Sebastian Nanz (Ed.)

THE FUTURE OF SOFTWARE ENGINEERING

Springer

The Future of Software Engineering

Sebastian Nanz
Editor

The Future of Software Engineering

Springer

Editor
Dr. Sebastian Nanz
ETH Zürich
Department of Computer Science
Clausiusstr. 59
8092 Zürich
Switzerland
nanz@inf.ethz.ch

ISBN 978-3-642-15186-6 e-ISBN 978-3-642-15187-3
DOI 10.1007/978-3-642-15187-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2010937182

Cover design: KuenkelLopka GmbH

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

In just a few decades, the development of quality software has evolved into one of the critical needs of our society. Meeting these needs would not be possible without the principles, best practices, and theoretical and technological foundations developed within the field of software engineering. Despite many major achievements over the years, an even larger number of challenges have emerged, making it easy to predict a long and exciting future for this discipline.

The contributions in this book provide an insight into the nature of these challenges, and how they are rooted in past achievements. Written by some of the field's most prominent researchers and technologists, the articles reflect the views of the individual authors on whatever they feel is important for the future of software engineering. Hence a broad range of topics is treated, both from academic and industrial perspectives.

The origin of this book was the idea to organize an event for Bertrand Meyer's 60th birthday. The contributed articles are therefore also the record of a symposium entitled "The Future of Software Engineering (FOSE)", held at ETH Zurich on 22–23 November 2010 on the occasion. All of the speakers at the symposium have worked with Bertrand at various times, so that the themes found in their articles demonstrate the wide span of his research interests.

It is our pleasure to thank all speakers, who agreed without hesitation to contribute a talk and on many accounts even a full paper, despite their many other research activities and busy schedules in general. We also thank everybody who contributed to the organization of the symposium, in particular Claudia Günthart and all members of the Chair of Software Engineering at ETH Zurich.

We hope the symposium and book will provide food for thought, stimulate discussions, and inspire research along the avenues highlighted in the articles. We also hope that the future of software engineering will see many of the authors' ideas realized, contributing to a prosperous growth of the field.

August 2010 Sebastian Nanz

Table of Contents

Afterword

Some Future Software Engineering Opportunities and Challenges

Barry Boehm

University of Southern California,
Los Angeles, CA 90089-0781
boehm@usc.edu

Abstract. This paper provides an update and extension of a 2006 paper, "Some Future Trends and Implications for Systems and Software Engineering Processes," *Systems Engineering*, Spring 2006. Some of its challenges and opportunities are similar, such as the need to simultaneously achieve high levels of both agility and assurance. Others have emerged as increasingly important, such as the challenges of dealing with ultralarge volumes of data, with multicore chips, and with software as a service. The paper is organized around eight relatively surprise-free trends and two "wild cards" whose trends and implications are harder to foresee. The eight surprise-free trends are:

1. Increasing emphasis on rapid development and adaptability;
2. Increasing software criticality and need for assurance;
3. Increased complexity, global systems of systems, and need for scalability and interoperability;
4. Increased needs to accommodate COTS, software services, and legacy systems;
5. Increasingly large volumes of data and ways to learn from them;
6. Increased emphasis on users and end value;
7. Computational plenty and multicore chips;
8. Increasing integration of software and systems engineering;

The two wild-card trends are:

9. Increasing software autonomy; and
10. Combinations of biology and computing.

1 Introduction

Between now and 2025, the ability of organizations and their products, systems, and services to compete, adapt, and survive will depend increasingly on software. As is being seen in current products (automobiles, aircraft, radios) and services (financial, communications, defense), software provides both competitive differentiation and rapid adaptability to competitive change. It facilitates rapid tailoring of products and services to different market sectors, and rapid and flexible supply chain management. The resulting software-intensive systems face ever-increasing demands to provide safe, secure, and reliable systems; to provide competitive discriminators in the marketplace; to support the coordination of multi-cultural global enterprises; to enable

S. Nanz (ed.), *The Future of Software Engineering*,
DOI 10.1007/978-3-642-15187-3_1,

rapid adaptation to change; and to help people cope with complex masses of data and information. These demands will cause major differences in the processes currently used to define, design, develop, deploy, and evolve a diverse variety of software-intensive systems.

This paper is an update of one written in late 2005 called, "Some Future Trends and Implications for Systems and Software Engineering Processes." One of its predicted trends was an increase in rates of change in technology and the environment. A good way to calibrate this prediction is to identify how many currently significant trends that the 2005 paper failed to predict. These include:

- Use of multicore chips to compensate for the decrease in Moore's Law rates of microcircuit speed increase—these chips will keep on the Moore's Law curve of computing operations per second, but will cause formidable problems in going from efficient sequential software programs to efficient parallel programs [80];
- The explosion in sources of electronic data and ways to search and analyze them, such as search engines and recommender systems [1];
- The economic viability and growth in use of cloud computing and software as a service [36]; and
- The ability to scale agile methods up to 100-person Scrums of Scrums, under appropriate conditions [19].

The original paper identified eight relatively surprise-free future trends whose interactions presented significant challenges, and an additional two wild-card trends whose impact is likely to be large, whose likely nature and realizations are hard to predict. This paper has revised the eight "surprise-free" trends to reflect the new trends above, but it has left the two wild-card trends as having remained but less predictable.

2 Future Software Engineering Opportunities and Challenges

2.1 Increasing emphasis on rapid development and adaptability

The increasingly rapid pace of systems change discussed above translates into an increasing need for rapid development and adaptability in order to keep up with one's competition. A good example was Hewlett Packard's recognition that their commercial product lifetimes averaged 33 months, while their software development times per product averaged 48 months. This led to an investment in product line architectures and reusable software components that reduced software development times by a factor of 4 down to 12 months [48].

Another response to the challenge of rapid development and adaptability has been the emergence of agile methods [6,51,28,90]. Our original [20] analysis of these methods found them generally not able to scale up to larger products. For example, Kent Beck says in [6], "Size clearly matters. You probably couldn't run an XP (eXtreme Programming) project with a hundred programmers. Not fifty. Not twenty, probably. Ten is definitely doable."

However, over the last decade, several organizations have been able to scale up agile methods by using two layers of 10-person Scrum teams. This involves, among other things, having each Scrum team's daily stand-up meeting followed up by a daily stand-up meeting of the Scrum team leaders, and by up-front investments in an evolving system architecture. We have analyzed several of these projects and organizational initiatives in [19]; a successful example and a partial counterexample are provided next.

The *successful example* is provided by a US medical services company with over 1000 software developers in the US, two European countries, and India. The corporation was on the brink of failure, due largely to its slow, error-prone, and incompatible software applications and processes. A senior internal technical manager, expert in both safety-critical medical applications and agile development, was commissioned by top management to organize a corporate-wide team to transform the company's software development approach. In particular, the team was to address agility, safety, and Sarbanes-Oxley governance and accountability problems.

Software technology and project management leaders from all of its major sites were brought together to architect a corporate information framework and develop a corporate architected-agile process approach. The resulting Scrum of Scrums approach was successfully used in a collocated pilot project to create the new information framework while maintaining continuity of service in their existing operations.

Based on the success of this pilot project, the team members returned to their sites and led similar transformational efforts. Within three years, they had almost 100 Scrum teams and 1000 software developers using compatible and coordinated architected-agile approaches. The effort involved their customers and marketers in the effort. Expectations were managed via the pilot project. The release management approach included a 2–12 week architecting Sprint Zero, a series of 3–10 one-month development Sprints, a Release Sprint, and 1–6 months of beta testing; the next release Sprint Zero overlapped the Release Sprint and beta testing. Their agile Scrum approach involved a tailored mix of eXtreme Programming (XP) and corporate practices, 6–12 person teams with dedicated team rooms, and global teams with wiki and daily virtual meeting support—working as if located next-door. Figure 1 shows this example of the Architected Agile approach.

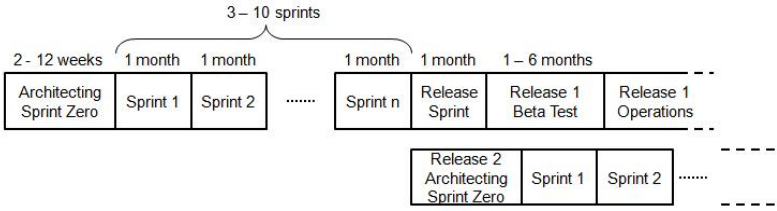

Fig. 1. Example of Architected Agile Process

Two of the other success stories had similar approaches. However, circumstances may require *different tailorings* of the architected agile approach. Another variant analyzed was an automated maintenance system that found its Scrum teams aligned

with different stakeholders whose objectives diverged in ways that could not be reconciled by daily standup meetings. The project recognized this and has evolved to a more decentralized Scrum-based approach, with centrifugal tendencies monitored and resolved by an empowered Product Framework Group (PFG) consisting of the product owners and technical leads from each development team, and the project systems engineering, architecting, construction, and test leads. The PFG meets near the end of an iteration to assess progress and problems, and to steer the priorities of the upcoming iteration by writing new backlog items and reprioritizing the product backlog. A few days after the start of the next iteration, the PFG meets again to assess what was planned vs. what was needed, and to make necessary adjustments. This has been working much more successfully.

2.2 Increasing Criticality and Need for Assurance

A main reason that products and services are becoming more software-intensive is that software can be more easily and rapidly adapted to change as compared to hardware. A representative statistic is that the percentage of functionality on modern aircraft determined by software increased to 80% by 2000 [46]. Although people's, systems', and organizations' dependency on software is becoming increasingly critical, the current major challenge in achieving system dependability is that dependability is generally not the top priority for software producers. In the words of the 1999 (U.S.) President's Information Technology Advisory Council (PITAC) Report, "The IT industry spends the bulk of its resources, both financial and human, on rapidly bringing products to market" [83].

This situation will likely continue until a major software-induced systems catastrophe similar in impact on world consciousness to the 9/11 World Trade Center catastrophe stimulates action toward establishing accountability for software dependability. Given the high and increasing software vulnerabilities of the world's current financial, transportation, communications, energy distribution, medical, and emergency services infrastructures, it is highly likely that such a software-induced catastrophe will occur between now and 2025.

Process strategies for highly dependable software-intensive systems and many of the techniques for addressing its challenges have been available for quite some time. A landmark 1975 conference on reliable software included papers on formal specification and verification processes; early error elimination; fault tolerance; fault tree and failure modes and effects analysis; testing theory, processes and tools; independent verification and validation; root cause analysis of empirical data; and use of automated aids for defect detection in software specifications and code [16]. Some of these were adapted from existing systems engineering practices; some were developed for software and adapted for systems engineering.

These have been used to achieve high dependability on smaller systems and some very large self-contained systems such as the AT&T telephone network [75]. Also, new strategies have been emerging to address the people-oriented and value-oriented challenges discussed in Section 2.1. These include the Personal and Team Software Processes [55,56], value/risk-based processes for achieving dependability objectives

[47,54], and value-based systems engineering processes such as Lean Development [95].

Many of the traditional assurance methods such as formal methods are limited in their speed of execution, need for scarce expert developers, and adaptability (often requiring correctness proofs to start over after a requirements change). More recently, some progress has been made in strengthening assurance methods and making them more adaptable. Examples are the use of the ecological concepts of "resilience" as a way to achieve both assurance and adaptability [52,61]; the use of more incremental assurance cases for reasoning about safety, security, and reliability [60]; and the development of more incremental correctness proof techniques [98].

2.2.1 An Incremental Development Process for Achieving Both Agility and Assurance

Simultaneously achieving high assurance levels and rapid adaptability to change requires new approaches to software engineering processes. Figure 2 shows a single increment of the incremental evolution portion of such a model, as presented in the [11] paper and subsequently adapted for use in several commercial organizations needing both agility and assurance. It assumes that the organization has developed:

- A best-effort definition of the system's steady-state capability;
- An incremental sequence of prioritized capabilities culminating in the steady-state capability; and
- A Feasibility Rationale providing sufficient evidence that the system architecture will support the incremental capabilities, that each increment can be developed within its available budget and schedule, and that the series of increments create a satisfactory return on investment for the organization and mutually satisfactory outcomes for the success-critical stakeholders.

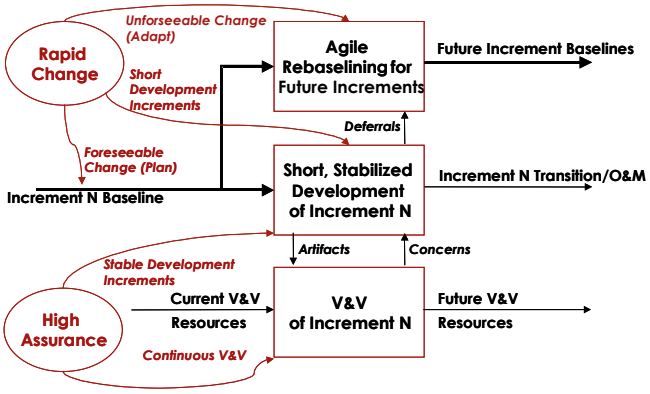

Fig. 2. The Incremental Commitment Spiral Process Model: Increment Activities

In *Balancing Agility and Discipline* [20], we found that rapid change comes in two primary forms. One is relatively predictable change that can be handled by the plan-driven Parnas strategy [79] of encapsulating sources of change within modules, so

that the effects of changes are largely confined to individual modules. The other is relatively unpredictable change that may appear simple (such as adding a "cancel" or "undo" capability [5]), but often requires a great deal of agile adaptability to rebaseline the architecture and incremental capabilities into a feasible solution set.

The need to deliver high-assurance incremental capabilities on short fixed schedules means that each increment needs to be kept as stable as possible. This is particularly the case for very large systems of systems with deep supplier hierarchies (often 6 to 12 levels), in which a high level of rebaselining traffic can easily lead to chaos. In keeping with the use of the spiral model as a risk-driven process model generator, the risks of destabilizing the development process make this portion of the project into a waterfall-like build-to-specification subset of the spiral model activities. The need for high assurance of each increment also makes it cost-effective to invest in a team of appropriately skilled personnel to continuously verify and validate the increment as it is being developed.

However, "deferring the change traffic" does not imply deferring its change impact analysis, change negotiation, and rebaselining until the beginning of the next increment. With a single development team and rapid rates of change, this would require a team optimized to develop to stable plans and specifications to spend much of the next increment's scarce calendar time performing tasks much better suited to agile teams.

The appropriate metaphor for these tasks is not a build-to-specification metaphor or a purchasing-agent metaphor but an adaptive "command-control-intelligence-surveillance-reconnaissance" (C2ISR) metaphor. It involves an agile team performing the first three activities of the C2ISR "Observe, Orient, Decide, Act" (OODA) loop for the next increments, while the plan-driven development team is performing the "Act" activity for the current increment. "Observing" involves monitoring changes in relevant technology and COTS products, in the competitive marketplace, in external interoperating systems and in the environment; and monitoring progress on the current increment to identify slowdowns and likely scope deferrals. "Orienting" involves performing change impact analyses, risk analyses, and tradeoff analyses to assess candidate rebaselining options for the upcoming increments. "Deciding" involves stakeholder renegotiation of the content of upcoming increments, architecture rebaselining, and the degree of COTS upgrading to be done to prepare for the next increment. It also involves updating the future increments' Feasibility Rationales to ensure that their renegotiated scopes and solutions can be achieved within their budgets and schedules.

A successful rebaseline means that the plan-driven development team can hit the ground running at the beginning of the "Act" phase of developing the next increment, and the agile team can hit the ground running on rebaselining definitions of the increments beyond. This model is similar to the ones used by the major cell phone software developers, who may run several concurrent teams phased to deliver new capabilities every 90 days.

2.3 Increased complexity, global systems of systems, and need for scalability and interoperability

The global connectivity provided by the Internet provides major economies of scale and network economies [4] that drive both an organization's product and process strategies. Location-independent distribution and mobility services create both rich new bases for synergetic collaboration and challenges in synchronizing activities. Differential salaries provide opportunities for cost savings through global outsourcing, although lack of careful preparation can easily turn the savings into overruns. The ability to develop across multiple time zones creates the prospect of very rapid development via three-shift operations, although again there are significant challenges in management visibility and control, communication semantics, and building shared values and trust. It also implies that collaborative activities such as Participatory Design [44] will require stronger human systems and software engineering process and skill support not only across application domains but also across different cultures.

A lot of work needs to be done to establish robust success patterns for global collaborative processes. Key challenges as discussed above include cross-cultural bridging; establishment of common shared vision and trust; contracting mechanisms and incentives; handovers and change synchronization in multi-time-zone development; and culture-sensitive collaboration-oriented groupware. Most software packages are oriented around individual use; just determining how best to support groups will take a good deal of research and experimentation. Within individual companies, such as IBM, corporate global collaboration capabilities have made collaborative work largely location-independent, except for large time-zone bridging inconveniences.

One collaboration process whose future applications niche is becoming better understood is open-source software development. Security experts tend to be skeptical about the ability to assure the secure performance of a product developed by volunteers with open access to the source code. Feature prioritization in open source is basically done by performers; this is generally viable for infrastructure software, but less so for competitive corporate applications systems and software. Proliferation of versions can be a problem with volunteer developers. But most experts see the current success of open source development for infrastructure products such as Linux, Apache, and Firefox as sustainable into the future.

Traditionally (and even recently for some forms of agile methods), systems and software development processes were recipes for standalone "stovepipe" systems with high risks of inadequate interoperability with other stovepipe systems. Experience has shown that such collections of stovepipe systems cause unacceptable delays in service, uncoordinated and conflicting plans, ineffective or dangerous decisions, and inability to cope with rapid change.

During the 1990's and early 2000's, standards such as ISO/IEC 12207 [58] and ISO/IEC 15288 [59] began to emerge that situated systems and software project processes within an enterprise framework. Concurrently, enterprise architectures such as the IBM Zachman Framework [99], RM-ODP [85], and the U.S. Federal Enterprise Framework [45], have been developing and evolving, along with a number of commercial Enterprise Resource Planning (ERP) packages.

These frameworks and support packages are making it possible for organizations to reinvent themselves around transformational, network-centric systems of systems. As discussed in [50], these are necessarily software-intensive systems of systems (SISOS), and have tremendous opportunities for success and equally tremendous risks of failure. Examples of successes have been Federal Express; Wal-Mart; and the U.S. Command, Control, Intelligence, Surveillance, and Reconnaissance (C2ISR) system in Iraq; examples of failures have been the Confirm travel reservation system, K-Mart, and the U.S. Advanced Automation System for air traffic control. ERP packages have been the source of many successes and many failures, implying the need for considerable risk/opportunity assessment before committing to an ERP-based solution.

Our work in supporting SISOS development programs has shown that the use of a risk-driven spiral process with early attention to SISOS risks and systems architecting methods [88] can avoid many of the SISOS development pitfalls [14,33]. A prioritized list of the top ten SISOS risks we have encountered includes several of the trends we have been discussing: (1) acquisition management and staffing, (2) requirements/architecture feasibility, (3) achievable software schedules, (4) supplier integration, (5) adaptation to rapid change, (6) systems and software quality factor achievability, (7) product integration and electronic upgrade, (8) software COTS and reuse feasibility, (9) external interoperability, and (10) technology readiness.

2.4 Increased needs to accommodate COTS, software services, and legacy systems

A 2001 ACM Communications editorial stated, “In the end—and at the beginning—it’s all about programming” [30]. Future trends are making this decreasingly true. Although infrastructure software developers will continue to spend most of their time programming, most application software developers are spending more and more of their time assessing, tailoring, and integrating commercial-off-the-shelf (COTS) products. And more recently, the COTS products need to be evaluated with respect to purchased software-as-a-service options.

The left side of Figure 3 illustrates the COTS trends for a longitudinal sample of small e-services applications going from 28% COTS-based in 1996-97 to 70% COTS-based in 2001-2002, plus a corroborative industry-wide 54% figure (the asterisk *) for COTS-based applications (CBAs) in the 2000 Standish Group survey [91,97]. COTS software products are particularly challenging to integrate. They are opaque and hard to debug. They are often incompatible with each other due to the need for competitive differentiation. They are uncontrollably evolving, averaging about to 10 months between new releases, and generally unsupported by their vendors after 3 subsequent releases. These latter statistics are a caution to organizations outsourcing applications with long gestation periods. In one case, we observed an outsourced application with 120 COTS products, 46% of which were delivered in a vendor-unsupported state [97].

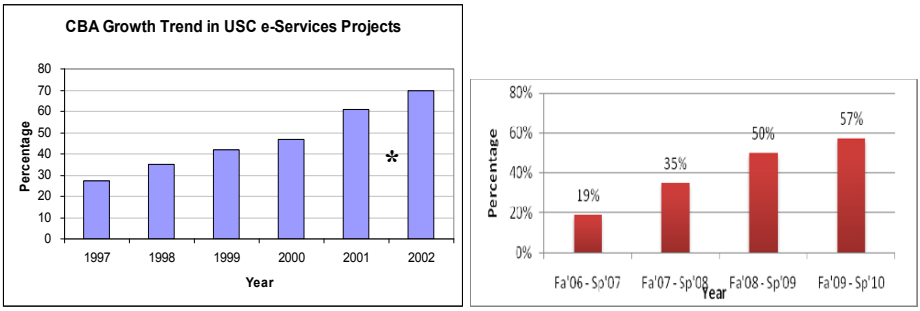

Fig. 3. COTS Usage Growth in USC E-Service Projects; Recent Purchased Software Services Growth

The right side of Figure 3 shows the corresponding recent growth in the use of purchased software services for a similar longitudinal sample of small e-services applications going from 19% in 2006-07 to 57% in 2009-10 Koolmanojwong, 2009]. Relative to COTS products, purchased software services have the advantages of eliminating the costs and failure modes of operating one's own server computers, low initial costs, accessibility across more types of user platforms, and having automatic upgrades. They have the disadvantages of automatic upgrades (no way to keep applications stable by declining upgrades), loss of exclusive control of one's data and speed of performance, and need for internet access.

Open source software, or an organization's reused or legacy software, is less opaque and less likely to go unsupported. But such software can also have problems with interoperability and continuing evolution. In addition, it often places constraints on a new application's incremental development, as the existing software needs to be decomposable to fit the new increments' content and interfaces. Across the maintenance life cycle, synchronized refresh of a large number of continually evolving COTS, open source, reused, and legacy software and hardware becomes a major additional challenge.

Legacy or Brownfield software is one area in which "it's all about programming" is still largely valid. Today and increasingly in the future, most large software-intensive system (SIS) developments will be constrained by the need to provide continuity of service while migrating their services away from poorly structured and documented legacy software applications. The International Data Corporation has estimated that there are 200 billion lines of such legacy codes in current operation [84]. Yet most SIS process models contain underlying assumptions that an application project can start from scratch in an unconstrained Greenfield approach to developing requirements and solutions. Frequently, such applications will turn out to be unusable when organizations find that there is no way to incrementally undo the Brownfield legacy software in order to phase in the new software.

Recently, several approaches have emerged for re-engineering the legacy software into a service-oriented architecture from which migration to newer software solutions can be accomplished. Examples are the IBM VITA approach [53], the SEI SMART approach [70], and application of the Incremental Commitment Spiral Model [12].

Besides programming, they require a good deal of up-front software systems engineering.

2.4.1 Systems and Software Engineering Process Implications

COTS and software-service economics generally makes sequential waterfall processes (in which the prespecified system requirements determine the capabilities) incompatible with COTS or service-based solutions (in which the COTS or service capabilities largely determine the requirements; a desired capability is not a requirement if you can't afford the custom solution that would provide it). Some initial software COTS and service-based applications (CBA) development processes are emerging. Some are based on composable process elements covering the major sources of CBA effort (assessment, tailoring, and glue code integration) [97]. Others are oriented around the major functions involved in software CBA's, such as the SEI EPIC process [2]. More recently, decision processes for choosing among COTS-based, services-based, agile, or hybrid processes in different software development situations have been developed and experimentally applied [63].

However, there are still major challenges for the future, such as processes for synchronized multi-COTS refresh across the life-cycle; processes for enterprise-level and systems-of-systems COTS, software services, open source, reuse, and legacy evolution; integrated hardware and software COTS-based system processes; and processes and techniques to compensate for shortfalls in multi-COTS and services usability, dependability, and interoperability. Such COTS and services-oriented shortfalls are another reason why emergent spiral or evolutionary processes are better matches to most future project situations than attempts to prespecify waterfall and V-model system and software development plans and schedules. For example, consider the following common development sequence:

1. Pick the best set of COTS products or services supporting the system objectives, and
2. Compensate for the selected COTS or services shortfalls with respect to the system objectives.

There is no way to elaborate task 2 (or other tasks depending on it) until task 1 is done, as different COTS or services selection will produce different shortfalls needing compensation.

2.5 Increasingly large volumes of data and ways to learn from them

The growth of the Internet has been matched by the growth of data sources connected to the Internet. Much of the data is in files protected by firewalls, passwords, or encryption, but a tremendous amount is available via people and organizations willing to share the data, or as part of internet transactions. The increasing economic attractiveness of cloud computing is creating huge data storage and processing complexes that can similarly be used to determine information of interest and economic value. The growing volume of this data has also been matched by the growing sophistication of the technology used to search and reason about the data.

Three primary technologies have been search engines, recommender systems, and general data mining techniques.

As one example the power of search engines, the largest numbers of search matches on Google found in the author's informal searching for instances of popular search terms were approximately 4.76 billion for "video," 3.74 billion for "time," 2.85 billion for "music," 2.57 billion for "life," 1.84 billion for "book," 1.59 billion for "game," and 1.54 billion for "food." Each of these searches was completed by Google in 0.15 to 0.22 seconds.

This example also points out the challenge of determining which matches out of a billion or so to show that are most likely to be of interest to the viewer or other interested parties. Clearly, some of the interested parties are vendors who would like to advertise their products to people who are interested in the search topic. Their willingness to pay for search providers to highlight their offerings is the main reason for the search providers' economic viability. A good summary of search engine technology is [24].

A good example of recommender systems is the capability developed and evolved by Amazon.com. This began as a way for Amazon to notify someone buying a book on Amazon of the other books most frequently bought by people that had bought the book they were buying. This is an example of collaborative filtering, which can be applied anywhere, as it does not require knowledge of the content of the items bought or accessed. Extensions of collaborative filtering to include item content have also been developed. Another type of recommender system asks users to identify a profile of likes, dislikes, and preferences either among example items or among attributes of items in a given area of interest (restaurants, vacations, music), and provides recommendations or services best satisfying the user's profile. A good summary of recommender systems is [1]. A good example of collaborative-filtering recommender technology is [64], summarizing the approach they used as part of the team that won the $1 million Netflix prize for improving Netflix's recommender system performance by more than 10%.

Data mining is a more general term for processes that extract patterns from data. It includes recommender systems and other techniques such as clustering algorithms that look for similarities among data elements; association rule learning (such as Amazon.com's rules for most-frequently-bought associated books); classification algorithms such as neural networks and Bayesian classification; and evaluation techniques such as regression and bootstrapping techniques.

Data mining techniques have become a widely-researched area within software engineering. In some cases, such as software cost estimation, the data are too scarce and imprecise for data mining techniques to have made much headway, but a recent issue of IEEE Software [76] includes a number of useful data mining results and prospects for stronger results in the future. These tend to have worked on sufficiently large data repositories in large companies or via the increasing volume of open source software. They include defect-prone module characterization; defect finding via inter-module consistency checking; detection of code churn hot-spots (and correlations with defect frequency) for test planning and remodularization around sources of change; plans-vs.-actuals tracking and dashboarding for early project risk analysis; and social network analysis of interpersonal interaction paths vs. integration failure rates. Another key to the general utility of such data mining results is the

availability of metadata on project size, domain, processes used, application criticality, etc., as results often vary significantly across different project types.

2.6 Increased emphasis on users and end value

A 2005 *Computerworld* panel on "The Future of Information Technology (IT)" indicated that usability and total ownership cost-benefits, including user inefficiency and ineffectiveness costs, are becoming IT user organizations' top priorities [3]. A representative quote from panelist W. Brian Arthur was "Computers are working about as fast as we need. The bottleneck is making it all usable." A recurring user-organization desire is to have technology that adapts to people rather than vice versa. This is increasingly reflected in users' product selection activities, with evaluation criteria increasingly emphasizing product usability and value added vs. a previous heavy emphasis on product features and purchase costs. Such trends ultimately will affect producers' product and process priorities, marketing strategies, and competitive survival.

Some technology trends strongly affecting usability and cost-effectiveness are increasingly powerful enterprise support packages, data access and mining tools, social networking applications, virtual reality applications, and increasingly powerful mobile computing and communications devices. Such products have tremendous potential for user value, but determining how they will be best configured will involve a lot of product experimentation, shakeout, and emergence of superior combinations of system capabilities. A further challenge is to track and accommodate changes in user capabilities and preferences; it increasingly appears that the next generation of users will have different strengths and weaknesses with respect to multitasking, attention span, and trial-and-error vs. thought-driven approaches to determining software solutions.

2.6.1 Systems and Software Engineering Process Implications

In terms of future systems and software process implications, the fact that the capability requirements for these products are emergent rather than prespecifiable has become the primary challenge. Not only do the users exhibit the IKIWISI (I'll know it when I see it) syndrome, but their priorities change with time. These changes often follow a Maslow need hierarchy, in which unsatisfied lower-level needs are top priority, but become lower priorities once the needs are satisfied [72]. Thus, users will initially be motivated by survival in terms of capabilities to process new workloads, followed by security once the workload-processing needs are satisfied, followed by self-actualization in terms of capabilities for analyzing the workload content for self-improvement and market trend insights once the security needs are satisfied. Chapter 1 of the recent Handbook of Human Systems Integration [21] summarizes the increased emphasis on human factors integration into systems engineering, and its state of progress in several large government organizations.

It is clear that requirements emergence is incompatible with past process practices such as requirements-driven sequential waterfall process models and formal programming calculi; and with process maturity models emphasizing repeatability

and optimization [81]. In their place, more adaptive [51] and risk-driven [9] models are needed. More fundamentally, the theory underlying software process models needs to evolve from purely reductionist "modern" world views (universal, general, timeless, written) to a synthesis of these and situational "postmodern" world views (particular, local, timely, oral) as discussed in [92]. A recent theory of value-based software engineering (VBSE) and its associated software processes [17] provide a starting point for addressing these challenges, and for extending them to systems engineering processes. More recently, Fred Brooks' book, *The Design of Design*, contains a framework and numerous insights and case studies on balancing the modern and postmodern approaches when designing artifacts or systems [23].

A book on VBSE approaches [7] contains further insights and emerging directions for VBSE processes. For example, the chapter on "Stakeholder Value Proposition Elicitation and Reconciliation" in the VBSE book [49] addresses the need to evolve from software products, methods, tools, and educated students strongly focused on individual programming performance to a focus on more group-oriented interdisciplinary collaboration. Negotiation of priorities for requirements involves not only participation from users and acquirers on each requirement's relative mission or business value, but also participation from systems and software engineers on each requirement's relative cost and time to develop and difficulty of implementation.

The aforementioned *Handbook of Human Systems Integration* [21] identifies a number of additional principles and guidelines for integrating human factors concerns into the systems engineering process. In particular, it identifies the need to elevate human factor concerns from a micro-ergonomics to a macro-ergonomics focus on organization, roles, responsibilities, and group processes of collective observation, orientation, decision-making, and coordinated action.

More recently, a major National Research Council study called Human-System Integration in the System Development Process [82] identified some of the inhibitors to effective human-system integration, including hardware-oriented system engineering and management guidance, practices, and processes. It recommended an early version of the Incremental Commitment Spiral Model to be discussed in Section 3 as a way to balance hardware, software, and human factors engineering activities, and a set of recommended research areas. Some of its software-related recommendations are:

- Conduct a research program with the goal of revolutionizing the role of end users in designing the system they will use;
- Conduct research to understand the factors that contribute to system resilience, the role of people in resilient systems, and how to design more resilient systems;
- Refine and coordinate the definition of a systems development process that concurrently engineers the system's hardware, software, and human factors aspects, and accommodates the emergence of HSI requirements, such as the incremental commitment model;
- Research and develop shared representations to facilitate communication across different disciplines and life cycle phases;
- Research and develop improved methods and testbeds for systems of systems HSI; and

– Research and develop improved methods and tools for integrating incompatible legacy and external-system user interfaces.

These have led to several advanced environments and practices for stimulating collaborative cross-discipline innovation support. A summary of some of these is provided in [69]. It identified a number of critical success factors, such as including responsible play, focusing on team rewards, using both science and art, making it OK to fail, making it not-OK to not-fail, and competitive multi-sourcing.

2.7 Computational Plenty and Multicore Chips

As discussed in Section 1, the use of multicore chips to compensate for the decrease in Moore's Law rates of microcircuit speed increase will keep computing processor technology on the Moore's Law curve of computing operations per second, but will cause formidable problems in going from efficient sequential software programs to efficient parallel programs [80]. The supercomputer field has identified some classes of applications that can be relatively easily parallelized, such as computational fluid dynamics, weather prediction, Monte Carlo methods for modeling and simulation sensitivity analysis, parallel searching, and handling numerous independently-running programs in cloud computing. But for individual sequential programs, computations that need to wait for the results of other computations cannot proceed until the other computations finish, often leaving most of the processors unable to do useful work. In some cases, sequential programs will run more slowly on a multicore processor than on a single-core processor with comparable circuit speed. General solutions such as parallel programming languages (Patterson lists 50 attempts), optimizing compilers, and processor design can help somewhat, but the fundamental problems of sequential dependencies cannot be simply overcome. Two good recent sources of information on multicore technology and programming practices are the March 2010 special issue of IEEE Computer [57] and the summary of key multicore Internet resources in [39].

However, besides processors, the speed, reliability, and cost of other information technologies such as data storage, communications bandwidth, display resolution, and mobile device capabilities and power consumption continue to increase. This computational plenty will spawn new types of platforms (smart dust, smart paint, smart materials, nanotechnology, micro electrical-mechanical systems: MEMS), and new types of applications (sensor networks, conformable or adaptive materials, human prosthetics). These will present process-related challenges for specifying their configurations and behavior; generating the resulting applications; verifying and validating their capabilities, performance, and dependability; and integrating them into even more complex systems of systems.

Besides new challenges, then, computational plenty will enable new and more powerful process-related approaches. It will enable new and more powerful self-monitoring software and computing via on-chip co-processors for assertion checking, trend analysis, intrusion detection, or verifying proof-carrying code. It will enable higher levels of abstraction, such as pattern-oriented programming, multi-aspect oriented programming, domain-oriented visual component assembly, and

programming by example with expert feedback on missing portions. It will enable simpler brute-force solutions such as exhaustive case evaluation vs. complex logic.

It will also enable more powerful software, hardware, human factors, and systems engineering tools that provide feedback to developers based on domain knowledge, construction knowledge, human factors knowledge, systems engineering knowledge, or management knowledge. It will enable the equivalent of seat belts and air bags for user-programmers. It will support show-and-tell documentation and much more powerful system query and data mining techniques. It will support realistic virtual game-oriented systems and software engineering education and training. On balance, the added benefits of computational plenty should significantly outweigh the added challenges.

2.8 Increasing Integration of Software and Systems Engineering

Several trends have caused systems engineering and software engineering to initially evolve as largely sequential and independent processes. First, systems engineering began as a discipline for determining how best to configure various hardware components into physical systems such as ships, railroads, or defense systems. Once the systems were configured and their component functional and interface requirements were precisely specified, sequential external or internal contracts could be defined for producing the components. When software components began to appear in such systems, the natural thing to do was to treat them sequentially and independently as Computer Software Configuration Items.

Second, the early history of software engineering was heavily influenced by a highly formal and mathematical approach to specifying software components, and a reductionist approach to deriving computer software programs that correctly implemented the formal specifications. A "separation of concerns" was practiced, in which the responsibility of producing formalizable software requirements was left to others, most often hardware-oriented systems engineers. Some example quotes illustrating this approach are:

- "The notion of 'user' cannot be precisely defined, and therefore has no place in computer science or software engineering," E. W. Dijkstra, panel remarks, ICSE 4, 1979 [38].
- "Analysis and allocation of the system requirements is not the responsibility of the software engineering group but is a prerequisite for their work," CMU-SEI Software Capability Maturity Model, version 1.1, 1993 [81].

As a result, a generation of software engineering education and process improvement goals were focused on reductionist software development practices that assumed that other (mostly non-software people) would furnish appropriate predetermined requirements for the software.

Third, the business practices of contracting for components were well worked out. Particularly in the government sector, acquisition regulations, specifications, and standards were in place and have been traditionally difficult to change. The path of least resistance was to follow a "purchasing agent" metaphor and sequentially specify requirements, establish contracts, formulate and implement solutions, and use the

requirements to acceptance-test the solutions [93,94]. When requirements and solutions were not well understood or changing rapidly, knowledgeable systems and software engineers and organizations could reinterpret the standards to operate more flexibly, concurrently and pragmatically and to produce satisfactory systems [25,89]. But all too frequently, the sequential path of least resistance was followed, leading to the delivery of obsolete or poorly-performing systems.

As the pace of change increased and systems became more user-intensive and software-intensive, serious strains were put on the sequential approach. First, it was increasingly appreciated that the requirements for user-intensive systems were generally not prespecifiable in advance, but emergent with use. This undermined the fundamental assumption of sequential specification and implementation.

Second, having people without software experience determine the software specifications often made the software much harder to produce, putting software even more prominently on the system development's critical path. Systems engineers without software experience would minimize computer speed and storage costs and capacities, which causes software costs to escalate rapidly [8]. They would choose best-of-breed system components whose software was incompatible and time-consuming to integrate. They would assume that adding more resources would speed up turnaround time or software delivery schedules, not being aware of slowdown phenomena such as multiprocessor overhead [8] or Brooks' Law (adding more people to a late software project will make it later) [22].

Third, software people were recognizing that their sequential, reductionist processes were not conducive to producing user-satisfactory software, and were developing alternative software engineering processes (evolutionary, spiral, agile) involving more and more systems engineering activities. Concurrently, systems engineering people were coming to similar conclusions about their sequential, reductionist processes, and developing alternative "soft systems engineering" processes (e.g., [25]), emphasizing the continuous learning aspects of developing successful user-intensive systems. Similarly, the project management field is undergoing questioning about its underlying specification-planning-execution-control theory being obsolete and needing more emphasis on adaptation and value generation [65].

2.8.1 Systems and Software Engineering Process Implications

Many commercial organizations have developed more flexible and concurrent development processes [96]. Also, recent process guidelines and standards such as the Integrated Capability Maturity Model (CMMI) [27], ISO/IEC 12207 for software engineering [58], and ISO/IEC 15288 for systems engineering [59] emphasize the need to integrate systems and software engineering processes, along with hardware engineering processes and human engineering processes. They emphasize such practices as concurrent engineering of requirements and solutions, integrated product and process development, and risk-driven vs. document-driven processes. New process milestones enable effective synchronization and stabilization of concurrent processes [10,66].

However, contractual acquisition processes still lag behind technical processes. Many organizations and projects develop concurrent and adaptive development

processes, only to find them frustrated by progress payments and award fees emphasizing compliance with sequential document-driven deliverables. More recently, though, corporate and professional organizations have been integrating their software and systems engineering activities (e.g., Systems and Software Consortium, Inc., Systems and Software Technology Conference, Practical Systems and Software Measurement. A number of software engineering methods and tools have been extended to address systems engineering, such as the extension of the Unified Modeling Language into the Systems Modeling Language [78]. Recent software engineering Body of Knowledge compendia such as the Graduate Software Engineering 2009 Curriculum Guidelines [86] supported by ACM, IEEE, and INCOSE have strongly integrated software and systems engineering. A similar effort in the systems engineering area is currently underway. And software process models such as the spiral model have been extended to integrate software and systems engineering, such as the Incremental Commitment Spiral Model to be discussed in Section 3.

2.9 Wild Cards: Autonomy and Bio-Computing

"Autonomy" covers technology advancements that use computational plenty to enable computers and software to autonomously evaluate situations and determine best-possible courses of action. Examples include:

- Cooperative intelligent agents that assess situations, analyze trends, and cooperatively negotiate to determine best available courses of action;
- Autonomic software that uses adaptive control techniques to reconfigure itself to cope with changing situations;
- Machine learning techniques that construct and test alternative situation models and converge on versions of models that will best guide system behavior; and
- Extensions of robots at conventional-to-nanotechnology scales empowered with autonomy capabilities such as the above.

Combinations of biology and computing include:

- Biology-based computing, that uses biological or molecular phenomena to solve computational problems beyond the reach of silicon-based technology, and
- Computing-based enhancement of human physical or mental capabilities, perhaps embedded in or attached to human bodies or serving as alternate robotic hosts for (portions of) human bodies.

Examples of books describing these capabilities are Kurzweil's *The Age of Spiritual Machines* [68] and Drexler's books *Engines of Creation* and *Unbounding the Future: The Nanotechnology Revolution* [40,41]. They identify major benefits that can potentially be derived from such capabilities, such as artificial labor, human shortfall compensation (the five senses, healing, life span, and new capabilities for enjoyment or self-actualization), adaptive control of the environment, or redesigning the world to avoid current problems and create new opportunities.

On the other hand, these books and other sources such as Dyson's *Darwin Among the Machines: The Evolution of Global Intelligence* [43] and Joy's article, "Why the

Future Doesn't Need Us" [62], and Crichton's bio/nanotechnology novel *Prey* [31], identify major failure modes that can result from attempts to redesign the world, such as loss of human primacy over computers, overempowerment of humans, and irreversible effects such as plagues or biological dominance of artificial species. From a software process standpoint, processes will be needed to cope with autonomy software failure modes such as undebuggable self-modified software, adaptive control instability, interacting agent commitments with unintended consequences, and commonsense reasoning failures.

As discussed in Dreyfus and Dreyfus' *Mind Over Machine* [42], the track record of artificial intelligence predictions shows that it is easy to overestimate the rate of AI progress. But a good deal of AI technology is usefully at work today and, as we have seen with the Internet and World Wide Web, it is also easy to underestimate rates of IT progress as well. It is likely that the more ambitious predictions above will not take place by 2025, but it is more important to keep both the positive and negative potentials in mind in risk-driven experimentation with emerging capabilities in these wild-card areas between now and 2025.

3 A Scalable Spiral Process Model for 21st Century Systems and Software

3.1 21st Century System and Software Development and Evolution Modes

In the next ten to twenty years, several 21st century system and software development and evolution modes will have emerged as the most cost-effective ways to develop needed capabilities in the context of the trends discussed in Section 2. The four most common modes are likely to be exploratory development of unprecedented capabilities, business model-based user programming, hardware and software product lines, and network-centric systems of systems. Each is discussed below, along with the primary processes that will most likely best fit their situations.

Exploratory development processes will continue to be used for new products in mainstream organizations and in new areas such as nanotechnology, advanced biotechnology and robotics, virtual reality, and cooperative agent-based systems. They will still focus on highly flexible processes for skill-intensive rapid prototyping. But pressures for rapid transition from prototype to fielded product will increase the emphasis on the concept development phase to meet criteria for demonstrating that the new concepts can be made sufficiently robust, scalable, and cost-effectively producible. The process and associated product capabilities will also need to be selectively open in order to support open-innovation forms of collaborative development with other companies providing complementary capabilities [26,69].

Business model-based user programming will expand its scope to continue to address the need to produce more and more software capabilities by enabling them to be produced directly by users, as with spreadsheet programs, computer-aided design and manufacturing (CAD/CAM) and website development and evolution. Much of the expanded scope will be provided by better-integrated and more tailorable Enterprise Resource Planning (ERP) COTS packages. As discussed in Section 2.7,

computational plenty and increased domain understanding will enable more powerful, safer, and easier-to-use user programming capabilities such as programming-by-example with expert-system feedback on missing portions. Larger extensions to the ERP framework may be carried out by in-house software development, but specialty-houses with product-line-based solutions will become an increasingly attractive outsourcing solution.

General web-based user programming was just emerging into significance in 2005, and has rapidly burgeoned in the subsequent five years. The emergence of new mass-collaboration platforms such as YouTube, Facebook, the iPhone, and computing clouds has created an open marketplace for composable applications and services, and a software engineering area called opportunistic system development [77]. Although there are still composability challenges among these applications and services, technology is emerging to address them [13].

Hardware and software product lines on the hardware side will increasingly include product lines for transportation, communications, medical, construction, and other equipment. On the software side, they will increasingly include product lines for business services, public services, and information infrastructure. Compared to current product lines in these areas, the biggest challenges will be the increasing rates of change and decreasing half-lives of product line architectures, and the increasing proliferation of product line variabilities caused by globalization.

Network-centric systems of systems. As discussed in Section 2.6, similar challenges are being faced by organizations in the process of transforming themselves from collections of weakly coordinated, vertically integrated stovepipe systems into seamlessly interoperable network-centric systems of systems (NCSOS). The architectures of these NCSOS are highly software-intensive and, as with the product line architectures above, need to be simultaneously robust, scalable, and evolvable in flexible but controllable ways. In section 3.2, we describe an emerging scalable spiral process model for developing and evolving 21st century product lines and NCSOS.

3.2 Overview of the Incremental Commitment Spiral Model

Based on our experiences in adapting the spiral model to the development of software-intensive systems of systems representative of the 21st century trends discussed above, we have been converging on a scalable spiral process model that has shown in several implementations to date to scale well and help projects avoid many sources of project failure, from small e-services applications [15] to superlarge defense systems of systems [33], and multi-enterprise supply chain management systems.

A view of the Incremental Commitment Spiral Model is shown in Figure 4. As with the original spiral model, its expanding spirals reflect increasing cumulative levels of system understanding, cost, development time, product detail and process detail. These do not expand uniformly, but as a function of the relative risks of doing too much or too little of product and process definition. Thus, valuation and selection of COTS products may be the highest-risk item and receive most of the early effort, or it might be prototyping of user interfaces, operational concept scenarios, or alternative vehicle platform configurations.

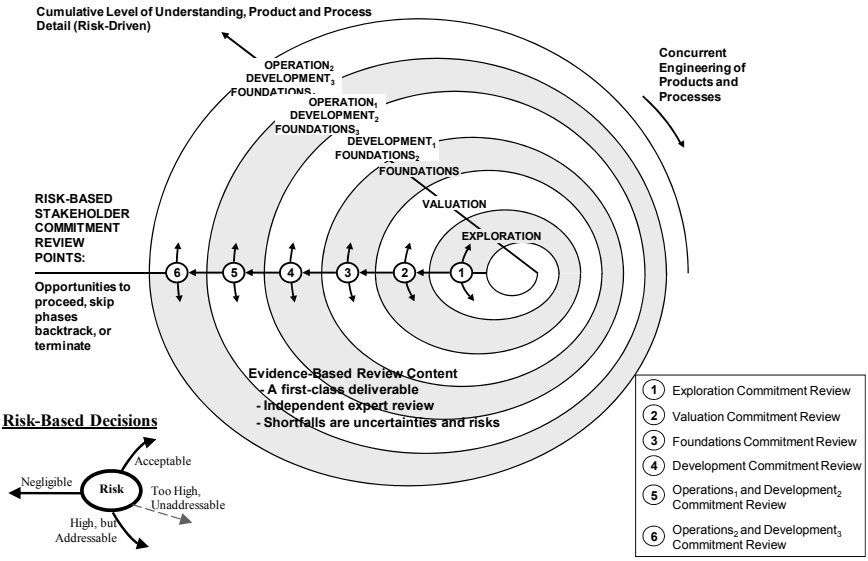

Fig. 4. The Incremental Commitment Spiral Model

Each spiral will be concurrently rather than sequentially addressing requirements and solutions; products and processes; hardware, software and human factors aspects; and business case analysis of alternative product configurations or product line investments. All of this concurrency is synchronized and stabilized by having the development team collaborate in producing not only artifacts, but also evidence of their combined feasibility. This evidence is then assessed at the various stakeholder commitment decision milestones by independent experts, and any shortfalls in evidence are considered as uncertainties or probabilities of loss, which when multiplied by the relative or absolute size of the prospective loss, becomes its level of Risk Exposure. Any such significant risks should then be addressed by a risk mitigation plan.

The stakeholders then consider the risks and risk mitigation plans, and decide on a course of action. If the risks are acceptable and will covered by risk mitigation plans, the project would proceed into the next spiral. If the risks are high but addressable, the project would remain in the current spiral until the risks are resolved (e.g., working out safety cases for a safety-critical system, or producing acceptable versions of missing risk mitigation plans). If the risks are negligible (e.g., finding at the end of the Exploration spiral that the solution can be easily produced via an already-owned COTS package which has been successfully used to produce more complex applications), there would be no need to perform a Valuation and a Foundations spiral, and the project could go straight into Development. If the risk is too high and unaddressable (e.g., the market window for such a product has already closed), the project should be terminated or rescoped, perhaps to address a different market sector whose market window is clearly sufficiently open. This outcome is shown by the dotted line "going into the third dimension" in the Risk-Based Decisions figure at the

lower left of Figure 4, but is not visible for simplicity on the numbered circles in the larger spiral diagram.

The Development spirals after the first Development Commitment Review follow the three-team incremental development approach for achieving both agility and assurance shown in Figure 2 and discussed in Section 2.2.1.

3.2.1 Other Views of the Incremental Commitment Spiral Model (ICSM)

Figure 5 presents an updated view of the ICSM life cycle process recommended in the National Research Council "Human-System Integration in the System Development Process" study [82]. It was called the Incremental Commitment Model (ICM) in the study, and given the study's sponsorship by the U.S. Department of Defense (DoD), also showed the DoD Instruction 5000.02 phases and milestones along with their generic ICSM counterparts.

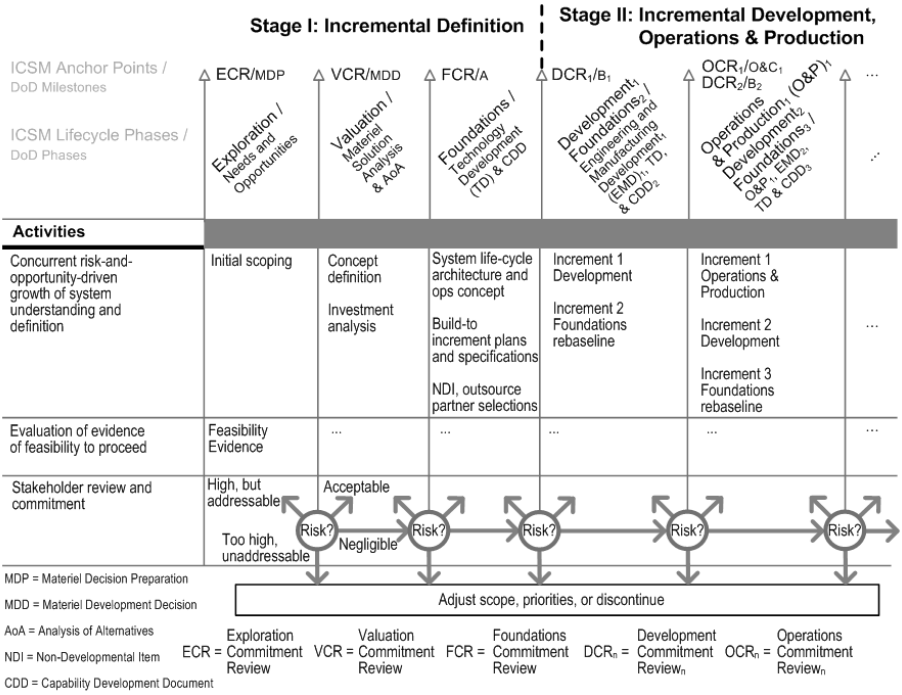

Fig. 5. Phased View of the Generic Incremental Commitment Spiral Model Process

The ICSM builds on the strengths of current process models: early verification and validation concepts in the V-model, concurrency concepts in the Concurrent Engineering model, lighter-weight concepts in the Agile and Lean models, risk-driven concepts in the spiral model, the phases and anchor points in the Rational Unified Process (RUP) [67,10], and recent extensions of the spiral model to address SoS capability acquisition [18].

In comparison to the software-intensive RUP (the closest widely-used predecessor to the ICM), the ICM also addresses hardware and human factors integration. It extends the RUP phases to cover the full system life cycle: an Exploration phase precedes the RUP Inception phase, which is refocused on valuation and investment analysis. The RUP Elaboration phase is refocused on Foundations (a term based on the [88] approach to Systems Architecting, describing concurrent development of requirements, architecture, and plans as the essential foundations for Engineering and Manufacturing Development), to which it adds feasibility evidence as a first-class deliverable. The RUP Construction and Transition phases are combined into Development; and an additional Operations phase combines operations, production, maintenance, and phase-out. Also, the names of the milestones are changed to emphasize that their objectives are to ensure stakeholder commitment to proceed to the next level of resource expenditure based on a thorough feasibility and risk analysis, and not just on the existence of a set of system objectives and a set of architecture diagrams. Thus, the RUP Life Cycle Objectives (LCO) milestone is called the Foundations Commitment Review (FCR) in the ICM and the RUP Life Cycle Architecture (LCA) milestone is called the Development Commitment Review (DCR).

The top row of Activities in Figure 5 indicates that a number of system aspects are being concurrently engineered at an increasing level of understanding, definition, and development. The most significant of these aspects are shown in Figure 6, an extension of a similar view of concurrently engineered software projects developed as part of the RUP [67].

As with the RUP version, it should be emphasized that the magnitude and shape of the levels of effort will be risk-driven and likely to vary from project to project. In particular, they are likely to have mini risk/opportunity-driven peaks and valleys, rather than the smooth curves shown for simplicity in Figure 6. The main intent of this view is to emphasize the necessary concurrency of the primary success-critical activities shown as rows in Figure 6. Thus, in interpreting the Exploration column, although system scoping is the primary objective of the Exploration phase, doing it well involves a considerable amount of activity in understanding needs, envisioning opportunities, identifying and reconciling stakeholder goals and objectives, architecting solutions, life cycle planning, evaluating alternatives, and negotiating stakeholder commitments.

For example, if one were exploring the initial scoping of a new medical device product line, one would not just interview a number of stakeholders and compile a list of their expressed needs into a requirements specification. One would also *envision and explore opportunities* for using alternative technologies, perhaps via competitive prototyping. In the area of *understanding needs*, one would determine relevant key performance parameters, scenarios, and evaluation criteria for evaluating the prototypers' results. And via the prototypes, one would explore alternative *architectural concepts* for developing, producing, and evolving the medical device product line; *evaluate* their relative feasibility, benefits, and risks for stakeholders to review; and if the risks and rewards are acceptable to the stakeholders, to *negotiate commitments* of further resources to proceed into a Valuation phase with a clearer understanding of what level of capabilities would be worth exploring as downstream requirements.

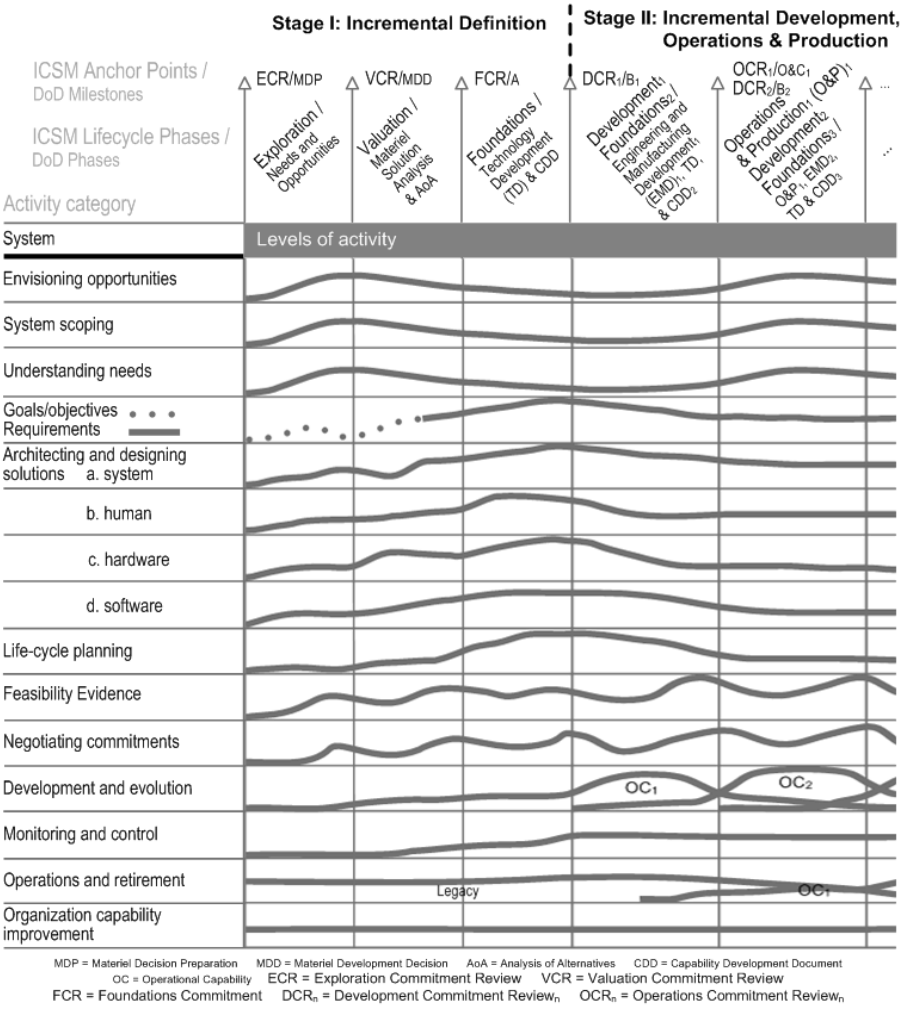

Fig. 6. ICSM Activity Categories and Level of Effort

Figure 6 indicates that a great deal of concurrent activity occurs within and across the various ICM phases, all of which needs to be synchronized and stabilized (a best-practice term taken from *Microsoft Secrets* [35] to keep the project under control. To make this concurrency work, the evidence-based anchor point milestone reviews are used to synchronize, stabilize, and risk-assess the ensemble of artifacts at the end of each phase. Each of these anchor point milestone reviews, labeled at the top of Figure 6, is focused on developer-produced and expert-reviewed *evidence*, instead of individual PowerPoint charts and Unified Modeling Language (UML) diagrams with associated assertions and assumptions, to help the key stakeholders determine the next level of commitment.

The review processes and use of independent experts are based on the highly successful AT&T Architecture Review Board procedures described in [71]. Figure 7 shows the content of the Feasibility Evidence Description. Showing feasibility of the concurrently-developed elements helps synchronize and stabilize the concurrent activities.

Feasibility Evidence Description Content

Evidence *provided by developer* and *validated by independent experts* that if the system is built to the specified architecture, it will:

- **Satisfy the requirements: capability, interfaces, level of service, and evolution**
- **Support the operational concept**
- **Be buildable within the budgets and schedules in the plan**
- **Generate a viable return on investment**
- **Generate satisfactory outcomes for all of the success-critical stakeholders**
- **Resolve all major risks by risk management plans**
- **Serve as basis for stakeholders' commitment to proceed**

Fig. 7. Feasibility Evidence Description Content

The Operations Commitment Review (OCR) is different, in that it addresses the often much higher operational risks of fielding an inadequate system. In general, stakeholders will experience a factor of two-to-ten increase in commitment level in going through the sequence of ECR to DCR milestones, but the increase in going from DCR to OCR can be much higher. These commitment levels are based on typical cost profiles across the various stages of the acquisition life-cycle.

3.2.2 Underlying ICSM Principles

At least as important as the diagrams depicting the ICSM views are its *four underlying principles*. If a project just follows the diagrams without following the principles (as often happened with the original spiral model), the project will have a serious risk of failure. The four principles are:

1. *Stakeholder value-based system definition and evolution*. If a project fails to include success-critical stakeholders such as end-users, maintainers, or suppliers, these stakeholders will frequently feel little commitment to the project and either underperform or refuse to use the results.
2. *Incremental commitment and accountability*. If success-critical stakeholders are not accountable for their commitments, they are likely to be drawn away to other pursuits when they are most needed.
3. *Concurrent system and software definition and development*. If definition and development of requirements and solutions; hardware, software, and human factors; or product and process definition are done sequentially, the project is likely both to go more slowly, and to make early, hard-to-undo commitments that cut off the best options for project success.

4. *Evidence and risk-based decision making.* If key decisions are made based on assertions, vendor literature, or meeting an arbitrary schedule without access to evidence of feasibility, the project is building up risks. And in the words of Tom Gilb, "If you do not actively attack the risks, the risks will actively attack you."

3.2.3 Model Experience to Date

During the National Research Council Human-Systems Integration study, it was found that the ICSM processes and principles corresponded well with best commercial practices. A good example documented in the study showed its application to a highly successful commercial medical infusion pump development [82], Chapter 5. A counterpart well-documented successful government-acquisition project using its principles was the CCPDS-R project described in Appendix D of [89].

A further source of successful projects that have applied the ICSM principles is the annual series of Top-5 software-intensive systems projects published in *CrossTalk* [32]. The "Top-5 Quality Software Projects" were chosen annually by panels of leading experts as role models of best practices and successful outcomes. Of the 20 Top-5 projects in 2002 through 2005, 16 explicitly used concurrent engineering; 14 explicitly used risk-driven development; and 15 explicitly used incrementally-committed evolutionary, iterative system growth, while additional projects gave indications of their partial use (The project summaries did not include discussion of stakeholder involvement). Evidence of successful results of stakeholder-satisficing can be found in the annual series of University of Southern California (USC) e-Services projects using the Win-Win Spiral model as described in [15]. Since 1998, over 50 user-intensive e-Services applications have used precursor and current versions of the ICSM to achieve a 92% success rate of on-time delivery of stakeholder-satisfactory systems. Its use on the ultralarge Future Combat Systems program enabled the sponsors to much better identify and deal with particularly the software-intensive program risks and identify improved courses of action [33].

A word of caution is that experiences to date indicate that the three teams' activities during evolutionary development are not as neatly orthogonal as they look in Figure 2. Feedback on development shortfalls from the V&V team either requires a response from the development team (early fixes will be less disruptive and expensive than later fixes), or deferral to a later increment, adding work and coordination by the agile team. The agile team's analyses and prototypes addressing how to accommodate changes and deferred capabilities need to draw on the experience and expertise of the plan-driven development team, requiring some additional development team resources and calendar time. Additional challenges arise if different versions of each increment are going to be deployed in different ways into different environments. The model has sufficient degrees of freedom to address such challenges, but they need to be planned for within the project's schedules and budgets.

4 Implications for 21st Century Enterprise Processes

In working with our commercial and aerospace Affiliates on how they can best evolve to succeed as 21st century enterprises, we have found several 20th century process-related institutions that need to be significantly rethought and reworked to contribute to success. We will discuss two leading examples below: acquisition practices and human relations. In the interest of brevity, some other important institutions needing rethinking and rework but not discussed in detail are continuous process improvement (repeatability and optimization around the past vs. adaptability and optimization around the future); supplier management (adversarial win-lose vs. team-oriented win-win); internal R&D strategies (core capability research plus external technology experimentation vs. full-spectrum self-invention); and enterprise integration (not-invented-here stovepipes vs. enterprise-wide learning and sharing).

4.1 Adaptive vs. Purchasing-Agent Acquisition

The 20th century purchasing agent or contracts manager is most comfortable with a fixed procurement to a set of prespecified requirements; selection of the least-cost, technically adequate supplier; and a minimum of bothersome requirements changes. Many of our current acquisition institutions—regulations, specifications, standards, contract types, award fee structures, reviews and audits—are optimized around this procurement model.

Such institutions have been the bane of many projects attempting to deliver successful systems in a world of emerging requirements and rapid change. The project people may put together good technical and management strategies to do concurrent problem and solution definition, teambuilding, and mutual-learning prototypes and options analyses. Then they find that their progress payments and award fees involve early delivery of complete functional and performance specifications. Given the choice between following their original strategies and getting paid, they proceed to get paid and marry themselves in haste to a set of premature requirements, and then find themselves repenting at leisure for the rest of the project (if any leisure time is available).

Build-to-specification contract mechanisms still have their place, but it is just for the stabilized increment development team in Figure 2. If such mechanisms are applied to the agile rebaselining teams, frustration and chaos ensues. What is needed for the three-team approach is separate contracting mechanisms for the three team functions, under an overall contract structure that enables them to be synchronized and rebalanced across the life cycle. Also needed are source selection mechanisms more likely to choose the most competent supplier, using such approaches as competitive exercises to develop representative system artifacts using the people, products, processes, methods, and tools in the offeror's proposal.

A good transitional role model is the CCPDS-R project described in [89]. Its US Air Force customer and TRW contractor (selected using a competitive exercise such as the one described above) reinterpreted the traditional defense regulations, specifications, and standards. They held a Preliminary Design Review: not a PowerPoint show at Month 4, but a fully validated architecture and demonstration of

the working high-risk user interface and networking capabilities at Month 14. The resulting system delivery, including over a million lines of software source code, exceeded customer expectations within budget and schedule.

Other good acquisition approaches are the Scandinavian Participatory Design approach [44], Checkland's Soft Systems Methodology [25], lean acquisition and development processes [96], and Shared Destiny-related contracting mechanisms and award fee structures [37,87]. These all reflect the treatment of acquisition using an adaptive-system metaphor rather than a purchasing-agent metaphor.

4.2 Human Relations

Traditional 20th century human relations or personnel organizations and processes tend to emphasize individual vs. team-oriented reward structures and monolithic career paths. These do not fit well with the team-oriented, diverse-skill needs required for 21st century success.

In *Balancing Agility and Discipline* [20], we found that plan-oriented people are drawn toward organizations that thrive on order. People there feel comfortable and empowered if there are clear policies and procedures defining how to succeed. On the other hand, agility people are drawn toward organizations that thrive on chaos. People there feel comfortable and empowered if they have few policies and procedures, and many degrees of freedom to determine how to succeed. In our USC Balancing Agility and Discipline Workshops, we found that most of our Affiliates had cultures that were strongly oriented toward one of these poles, with the challenge of evolving toward the other pole without losing the good features of their existing culture and staff. More recently, these workshops have surfaced the Architected Agile approach summarized in Section 2.1 [19].

The three-team approach presented in Section 2.2.1 provides a way for organizations to develop multiple role-oriented real or virtual skill centers with incentive structures and career paths focused both on excellence within one's preferred role and teamwork with the other contributors to success. Some other key considerations are the need for some rotation of people across team roles or as part of integrated product teams to avoid overspecialization, and the continual lookout for people who are good at all three team roles and are strong candidates for project-level or organization-level management or technical leadership careers.

A good framework for pursuing a human relations strategy for 21st century success is the People Capability Maturity Model [34]. Its process areas on participatory culture, workgroup development, competency development, career development, empowered workgroups, and continuous workforce innovation emphasize the types of initiatives necessary to empower people and organizations (such as the purchasing agents and departments discussed above) to cope with the challenges of 21st century system development. The P-CMM book also has several case studies of the benefits realized by organizations adopting the model for their human relations activities. The Collins *Good to Great* book [29] is organized around a stepwise approach characterizing the 11 outstanding performance companies' transformation into cultures having both an ethic of entrepreneurship and culture of discipline. It begins

with getting the right people and includes setting ambitious but achievable goals and constancy of purpose in achieving them.

5 Conclusions

The surprise-free and wild-card 21st century trends discussed in Section 2 provide evidence that significant changes in and integration of systems and software engineering processes will be needed for successful 21st century enterprises. Particularly important are changes that emphasize value generation and enable dynamic balancing of the agility, discipline, and scalability necessary to cope with the 21st century challenges of increasing rapid change, high dependability, and scalability to globally-integrated, software-intensive systems of systems.

Section 3 presents an incremental commitment spiral model (ICSM) process framework and set of product and process strategies for coping with these challenges. They are proving to be effective as we evolve them with our industry and government Affiliate organizations. The product strategies involve system and software architectures that encapsulate sources of rapid unpredictable change into elements developed by agile teams within a framework and set of elements developed by a plan-driven team. The process strategies involve stabilized increment development executed by the plan-driven team and verified and validated by a V&V team, along with concurrent agile, pro-active change assessment and renegotiation of stable plans and specifications for executing the next increment, as shown in Figure 2.

However, process and architecture strategies are only necessary and not sufficient conditions for enterprise success. Section 4 identifies and describes some of the complementary organizational initiatives that will be necessary to enable the product and process strategies to succeed. These include rethinking and reworking acquisition contracting practices, human relations, continuous process improvement, supplier management, internal R&D strategies, and enterprise integration.

As a bottom line, the future of software engineering will be in the hands of students learning software engineering over the next two decades. They will be practicing their profession well into the 2040's, 2050's and probably 2060's. The pace of change continues to accelerate, as does the complexity of the systems. This presents many serious, but exciting, challenges to software engineering education, including:

- Anticipating future trends (as in this book) and preparing students to deal with them;
- Capitalizing on information technology to enable the delivery of just-in-time and web-based education;
- Monitoring current principles and practices and separating timeless principles from outdated practices;
- Participating in leading-edge software engineering research and practice and incorporating the results into the curriculum;
- Packaging smaller-scale educational experiences in ways that apply to large-scale projects;
- Helping students learn how to learn, through state-of-the-art analyses, future-oriented educational games and exercises, and participation in research; and

– Offering lifelong learning opportunities for systems engineers who must update their skills to keep pace with the evolution of best practices, and individuals entering the software engineering field from outside disciplines, who need further education to make the transition successfully.

The annual ETH Zurich software engineering research and education summaries (e.g. [74,73]) are excellent examples of meeting these challenges.

References

1. Adomavicius, G., Tuzhilin, A.: Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. IEEE Transactions on Knowledge and Data Engineering 17(6) (June 2005) 734–749
2. Albert, C., Brownsword, L.: Evolutionary Process for Integrating COTS-Based Systems (EPIC): An Overview. CMU/SEI-2003-TR-009. Pittsburgh, PA: Software Engineering Institute (2002)
3. Anthes, G.: The Future of IT. Computerworld, (March 7, 2005) 27-36
4. Arthur, W. B.: Increasing Returns and the New World of Business. Harvard Business Review (July/August, 1996) 100-109
5. Bass, L., John, B.E.: Linking usability to software architecture patterns through general scenarios. Journal of Systems and Software 66 (3) (2003) 187-197
6. Beck, K.: Extreme Programming Explained, Addison Wesley (1999)
7. Biffl, S., Aurum, A., Boehm, B., Erdogmus, H., Gruenbacher, P. (eds.): Value-Based Software Engineering. Springer Verlag (2005)
8. Boehm, B.: Software Engineering Economics. Prentice Hall (1981)
9. Boehm, B.: A Spiral Model for Software Development and Enhancement. Computer (May, 1988) 61-72
10. Boehm, B.: Anchoring the Software Process. Software. (July, 1996) 73-82
11. Boehm, B.: Some Future Trends and Implications for Systems and Software Engineering Processes, Systems Engineering, Vol. 9, No. 1 (2006) 1-19
12. Boehm, B.: Applying the Incremental Commitment Model to Brownfield Systems Development, Proceedings, CSER 2009 (April 2009)
13. Boehm, B., Bhuta, J.: Balancing Opportunities and Risks in Component-Based Software Development, IEEE Software, November-December 2008, Volume 15, Issue 6, pp. 56-63.
14. Boehm, B., Brown, A.W., Basili, V., Turner, R.: Spiral Acquisition of Software-Intensive Systems of Systems. CrossTalk (May, 2004) 4-9
15. Boehm, B., Egyed, A., Kwan, J., Port, D., Shah, A., Madachy, R.: Using the WinWin Spiral Model: A Case Study, IEEE Computer (July 1998) 33-44
16. Boehm. B., Hoare, C.A.R. (eds.): Proceedings, 1975 International Conference on Reliable Software, ACM/IEEE (April, 1975)
17. Boehm, B., Jain, A.: An Initial Theory of Value-Based Software Engineering. In: Aurum, A., Biffl, S., Boehm, B., Erdogmus, H., Gruenbacher, P. (eds.): Value-Based Software Engineering, Springer Verlag (2005)
18. Boehm, B., Lane, J.: Using the Incremental Commitment Model to Integrate System Acquisition, Systems Engineering, and Software Engineering. CrossTalk (October 2007) 4-9
19. Boehm, B., Lane, J., Koolmanojwong, S., Turner, R.: Architected Agile Solutions for Software-Reliant Systems, Proceedings, INCOSE (2010)
20. Boehm, B., Turner, R.: Balancing Agility and Discipline. Addison Wesley (2004)
21. Booher, H.(ed.): Handbook of Human Systems Integration, Wiley (2003)

22. Brooks, F.: The Mythical Man-Month (2nd ed.). Addison Wesley (1995)
23. Brooks, F., The Design of Design, Addison Wesley, 2010.
24. Büttcher,S., Clarke, L., Cormack, G.: Information Retrieval: Implementing and Evaluating Search Engines. MIT Press (2010)
25. Checkland, P.: Systems Thinking, Systems Practice. 2nd ed., Wiley (1999)
26. Chesbrough, H.: Open Innovation, Harvard Business School Press (2003)
27. Chrissis, M., B., Konrad, M., Shrum, S.: CMMI. Addison Wesley (2003)
28. Cockburn, A.: Agile Software Development, Addison Wesley (2002)
29. Collins, J.: Good to Great, Harper Collins (2001)
30. Crawford, D.: Editorial Pointers. Comm. ACM (October, 2001) 5
31. Crichton, M.: Prey, Harper Collins (2002)
32. CrossTalk: Top Five Quality Software Projects, (January 2002), (July 2003), (July 2004), (September 2005) http://www.stsc.hill.af.mil/crosstalk
33. Crosson, S., Boehm, B.: Adjusting Software Life-Cycle Anchorpoints: Lessons Learned in a System of Systems Context, Proceedings SSTC 2009 (April 2009). Also TR USC-CSSE-2009-525
34. Curtis, B., Hefley, B., Miller, S.: The People Capability Maturity Model. Addison Wesley (2002)
35. Cusumano, M., Selby, R.: Microsoft Secrets. Harper Collins (1996)
36. Cusumano, M.: The Business of Software. New York: Free Press/Simon & Schuster (2004)
37. Deck, M., Strom, M., Schwartz, K.: The Emerging Model of Co-Development. Research Technology Management (December, 2001)
38. Dijkstra, E.: Panel remarks. Software Engineering: As It Should Be. ICSE 4 (September, 1979) – See also EWD 791 at http://www.cs.utexas/users/EWD
39. Doernhofer, M.: Multicore and Multithreaded Programming, ACM Sofware Engineering Notes (July 2010) 8-16.
40. Drexler, K.E.: Engines of Creation. Anchor Press (1986)
41. Drexler, K.E., Peterson, C., Pergamit, G.: Unbounding the Future: The Nanotechnology Revolution. William Morrow & Co. (1991)
42. Dreyfus, H., Dreyfus, S.: Mind over Machine. Macmillan (1986)
43. Dyson, G. B.: Darwin Among the Machines: The Evolution of Global Intelligence, Helix Books/Addison Wesley (1997)
44. Ehn, P. (ed.): Work-Oriented Design of Computer Artifacts, Lawrence Earlbaum Assoc. (1990)
45. FCIO (Federal CIO Council): A Practical Guide to Federal Enterprise Architecture, Version 1.0. (February, 2001)
46. Ferguson, J.: Crouching Dragon, Hidden Software: Software in DOD Weapon Systems, IEEE Software, vol. 18, no. 4, (July/August 2001) 105-107
47. Gerrard, P., Thompson, N.: Risk-Based E-Business Testing. Artech House (2002)
48. Grady, R.: Successful Software Process Improvement. Prentice Hall (1997)
49. Gruenbacher, P, Koszegi, S., Biffl, S., Stakeholder Value Proposition Elicitation and Reconciliation, in Aurum, A., Biffl, S., Boehm, B., Erdogmus, H., Gruenbacher, P. (eds.): Value-Based Software Engineering, Springer (2005)
50. Harned, D., Lundquist, J.: What Transformation Means for the Defense Industry. The McKinsey Quarterly, (November 3, 2003) 57-63
51. Highsmith, J.: Adaptive Software Development. Dorset House (2000)
52. Hollnagel, E., Woods, D., Leveson, N. (eds.): Resilience Engineering: Concepts and Precepts. Ashgate Publishing, (2006)
53. Hopkins, R., Jenkins, K.: Eating the IT Elephant: Moving from Greenfield Development to Brownfield. IBM Press (2008)

54. Huang, L.: A Value-Based Process Achieving Software Dependability. Proceedings, Software Process Workshop 2005 (May, 2005)
55. Humphrey, W.: Introduction to the Personal Software Process. Addison Wesley (1997)
56. Humphrey, W.: Introduction to the Team Software Process. Addison Wesley (2000)
57. IEEE Computer: Special Issue on Multicore Programming (March 2010)
58. ISO (International Standards Organization): Standard for Information Technology – Software Life Cycle Processes. ISO/IEC 12207 (1995)
59. ISO (International Standards Organization): Systems Engineering – System Life Cycle Processes. ISO/IEC 15288 (2002)
60. ISO (International Standards Organization): Systems and Software Engineering – Systems and Software Assurance – Part 2: Assurance Case (ISO/IEC 15026) (2009)
61. Jackson, S.: Architecting Resilient Systems. Wiley (2009)
62. Joy, B.: Why the Future Doesn't Need Us: Wired (April, 2000)
63. Koolmanojwong, S. The Incremental Commitment Model process patterns for rapid-fielding projects, Qualifying Exam Report (November 2009). Also TR USC-CSSE-2009-526
64. Koren, Y., Bell, R., Volinsky, C.: Matrix Factorization Techniques for Recommender Systems, Computer (August 2009) 30-37
65. Koskela, L., Howell, L.: The Underlying Theory of Project Management is Obsolete, Proceedings, PMI Research Conference (2002) 293-302
66. Kroll, P., Kruchten, P.: The Rational Unified Process Made Easy: A Practitioner's Guide to the Rational Unified Process. Addison Wesley (2003)
67. Kruchten, P.: The Rational Unified Process. Addison Wesley (1999)
68. Kurzweil, R.: The Age of Spiritual Machines. Penguin Books (1999)
69. Lane, J., Boehm, B., Bolas, M., Madni, A., Turner, R.: Critical Success Factors for Rapid, Innovative Solutions, Proceedings, ICSP 2010 (July 2010)
70. Lewis, G., Morris, E.J., Smith, D.B., Simanta, S.: SMART: Analyzing the Reuse Potential of Legacy Components on a Service-Oriented Architecture Environment, CMU/SEI-2008-TN-008 (2008)
71. Maranzano, J.F., Rozsypal, S.A., Zimmerman, G.H., Warnken, G.W., Wirth, P.E., Weiss, D.M.: Architecture reviews: Practice and experience. IEEE Software (March/April 2005) 34-43
72. Maslow, A.: Motivation and Personality. Harper and Row (1954)
73. Meyer, B., Mueller, P., Bay, T.: Software Engineering 2008, ETH Zurich Chair of Software Engineering (December 2008)
74. Meyer, B., Furia, C.: Software Engineering 2009, ETH Zurich Chair of Software Engineering (December 2009)
75. Musa, J.: Software Reliability Engineering. McGraw Hill (1999)
76. Nagappan, N., Zimmermann, T., Zeller, A. (eds.): Special Issue on Mining Software Archives, IEEE Software, (January/February 2009)
77. Ncube, C., Oberndorf, P., Kark, A. (eds.): Special Issue on Opportunistic System Development, IEEE Software (November/December 2008)
78. OMG (Object Management Group): OMG SysML v.1.2, http://www.sysml.org/specs.htm (June 2010)
79. Parnas, D.: Designing Software for Ease of Extension and Contraction. Transactions on Software Engineering, IEEE, SE-5, (1979)
80. Patterson, D.: The Trouble With Multicore, IEEE Spectrum, (July 2010) 28-32, 52-53.
81. Paulk, M., Weber, C., Curtis, B., Chrissis, M.: The Capability Maturity Model. Addison Wesley (1994)
82. Pew, R., Mavor, A. (eds.): Human-System Integration in the System development Process: A New Look, National Academies Press (2007)

83. PITAC (President's Information Technology Advisory Committee): Report to the President: Information Technology Research: Investing in Our Future (1999)
84. Price, H., Morley, J., Create, Apply, and Amplify: A Story of Technology Development, SEI Monitor (February 2009) 2
85. Putman, J.: Architecting with RM-ODP. Prentice Hall (2001)
86. Pyster, A., et al.: Graduate Software Engineering 2009 (GSwE2009) Curriculum Guidelines, Stevens Institute (September 2009)
87. Rational, Inc.: Driving Better Business with Better Software Economics, Rational Software Corp. (now part of IBM) (2001)
88. Rechtin, E.: Systems Architecting. Prentice Hall (1991)
89. Royce, W. E.: Software Project Management. Addison Wesley (1998)
90. Schwaber, K., Beedle, M.: Agile Software Development with Scrum, Prentice Hall (2002)
91. Standish Group: Extreme Chaos. http://www.standishgroup.com (2001)
92. Toulmin, S.: Cosmopolis. University of Chicago Press (1992)
93. U.S. Department of Defense, MIL-STD-1521B: Reviews and Audits (1985)
94. U.S. Department of Defense, DOD-STD-2167A: Defense System Software Development (1988)
95. Womack, J., Jones, D.: Lean Thinking: Banish Waste and Create Wealth in Your Corporation. Simon & Schuster (1996)
96. Womack, J. P., Jones, D. T., Roos, D.: The Machine that Changed the World: The Story of Lean production. Harper Perennial (1990)
97. Yang, Y., Bhuta, J., Port, D., Boehm, B.: Value-Based Processes for COTS-Based Applications. IEEE Software (July/August 2005) 54-62
98. Yin, X., Knight, J.: Formal Verification of Large Software Systems, Proceedings, NASA Formal Methods Symposium 2 (April 2010)
99. Zachman, J.: A Framework for Information Systems Architecture. IBM Systems Journal (1987)

Seamless Method- and Model-based Software and Systems Engineering

Manfred Broy

Institut für Informatik, Technische Universität München
D-80290 München Germany, broy@in.tum.de
http://wwwbroy.informatik.tu-muenchen.de

Abstract. Today engineering software intensive systems is still more or less handicraft or at most at the level of manufacturing. Many steps are done ad-hoc and not in a fully systematic way. Applied methods, if any, are not scientifically justified, not justified by empirical data and as a result carrying out large software projects still is an adventure. However, there is no reason why the development of software intensive systems cannot be done in the future with the same precision and scientific rigor as in established engineering disciplines. To do that, however, a number of scientific and engineering challenges have to be mastered. The first one aims at a deep understanding of the essentials of carrying out such projects, which includes appropriate models and effective management methods. What is needed is a portfolio of models and methods coming together with a comprehensive support by tools as well as deep insights into the obstacles of developing software intensive systems and a portfolio of established and proven techniques and methods with clear profiles and rules that indicate when which method is ready for application. In the following we argue that there is scientific evidence and enough research results so far to be confident that solid engineering of software intensive systems can be achieved in the future. However, yet quite a number of scientific research problems have to be solved.

Keywords: Formal methods, model based development

1 Motivation

Since more than four decades, extensive research in the foundations of software engineering accomplished remarkable results and a rich body of knowledge. Nevertheless the transfer to practice is slow – lagging behind the state of science and sometimes even not making much progress.

In the following we direct our considerations both to technical aspects of development and to management issues. In fact, a lot of the problems in software and systems engineering do not come from the software engineering techniques but rather from problems in project management as pointed out by Fred Brooks (see [2]). His book, “The Mythical Man-Month”, reflects experience in managing the development of OS/360 in 1964-65. His central arguments are that large projects suffer from management problems different in kind than small ones, due to division in labour, and

S. Nanz (ed.), *The Future of Software Engineering*,
DOI 10.1007/978-3-642-15187-3_2,

their critical need is the preservation of the conceptual integrity of the product itself. His central conclusions are that conceptual integrity can be achieved throughout by chief architects and implementation is achieved through well-managed effort.

The key question is what the decisive success factors for software and systems engineering technologies are. What we observe today is an enormous increase in functionality, size and complexity of software-based systems. Hence, one major goal in development is a reduction of complexity and a scaling up of methods to the size of the software. Moreover, technologies have to be cost effective. Techniques that contribute to the quality of the product but do not prove to be affordable are not helpful in practise.

2 Engineering Software Intensive Systems

Engineering is the systematic application of scientific principles and methods to the efficient and effective construction of useful structures and machines. Engineering of software intensive systems is still a challenge! The ultimate goal is to improve our abilities to manage the development and evolution of software intensive systems. Primary goals of engineering software intensive systems are suitable quality, low evolution costs and timely delivery.

2.1 Engineering and Modelling based on First Principles

To make sure that methods are helpful in engineering and really do address key issues, it is advisable to base development methods on principles and strategic goals. These principles are valid conclusions of the experiences gained in engineering software intensive systems. In the following we recapitulate a number of such principles and then discuss to what extent these principles can be backed up by specific methods.

One of the simple insights in software and systems engineering is, of course, that not only the way artefacts are described and also not just the methods that are applied are most significant for the quality of the development outcome. What is needed is a deep understanding of the engineering issues taking into account all kinds of often not explicitly stated quality requirements. As engineers, we are interested to be sure that our systems address the users' needs in a valid way and show required quality, for instance, that they are safe with a high probability and that during their lifecycle they can be evolved and adapted to the requirements in the future to come. In particular, evolving legacy software is one of the nightmares of software engineering. One major goal is keeping software evolvable. It is unclear to what extent available methods can help here.

The discipline of systems and software engineering has gathered a large amount of development principles and rules of best practice. Examples are principles like:

- separation of concerns
- stepwise refinement
- modularity and compositionality

- hierarchical decomposition
- standardized architecture and patterns
- abstraction
- information hiding
- rigor and formality
- generality – avoiding overspecification
- mitigation of risk
- incremental development
- anticipation of change
- scalability

Software Engineering "Maxims" say:

- Adding developers to a project will likely result in further delays and accumulated costs.
- Basic tension of software engineering is in trade-offs such as:
  - Better, cheaper, faster — pick any two!
  - Functionality, scalability, performance — pick any two!
- The longer a fault exists in software
  - the more costly it is to detect and correct,
  - the less likely it is to be properly corrected.
- Up to 70% of all faults detected in large-scale software projects are introduced in requirements and design.
- Insufficient communication and transparency in the development team will lead to project failure.
- Detecting the causes of faults early may reduce their resulting costs by a factor of 100 or more.

How can specific development methods evaluate these rules of thumb and support these principles? Some only address management issues.

2.2 From Principles to Methods, from Methods to Processes

Given principles, we may ask how to derive from known and proven methods and from methods development processes.

2.2.1 Key Steps in Software and Systems Engineering

In fact, looking at projects in practise we may identify the key activities in software and systems engineering. When studying projects and their success factors on the technical side, the latter prove to be always the same, namely, valid requirements, well worked out architectures both addressing the needs of the users as well as the application domain and an appropriate mapping onto technical solutions and adequate and proactive management. Constructive and analytical quality assurance is essential. However, only a part of it is verification not to forget validation of the requirements. An important issue that is not addressed enough in methods and techniques is comprehensibility and understandability. Formal description techniques are precise,

of course. But the significant goal is reduction of complexity and ease of understanding – this is why graphical description techniques seem to be so popular! Engineers, users, stakeholders have to understand the artefacts worked out during the development process. Often understanding is even more important then formality. Up to 60 % and more of the effort spent in software evolution and maintenance is code understanding. If a formal method precisely captures important properties, but if engineers are not able to understand it properly the way it is formulated then the method is not useful in practice.

2.2.2 Requirements Engineering

Gathering requirements based on collecting, structuring, formalizing, specifying, modelling are key activities. One of the big issues is, first of all, capturing and structuring valid requirements. IEEE standard 830-1998 mentions the following quality attributes of requirements documentation. Requirements specification and documentation has to be correct, unambiguous, complete, consistent, ranked for importance/stability, verifiable, modifiable, and traceable.

For effective requirements engineering we do not necessarily need formality of methods to begin with, since from the attributes listed above mainly consistency, unambiguity, and verifiability are supported directly by formal methods. A much better systematics for requirements is needed.

2.2.3 Architecture Design

Architecture aims at structuring software systems. Usually there is not just one architectural view onto a software system, but many related architectural viewpoints. A comprehensive architecture includes the most significant viewpoints. We give a short overview over viewpoints as they proved to be useful in automotive software and systems engineering:

- *Usage Process Viewpoint*: user processes and use cases,
- *Functional Viewpoint*: decomposition of systems into system function hierarchy, dependencies, functional specification of functions and their dependencies,
- *Logical component viewpoint*: decomposition of systems into data flow networks of logical system components, component hierarchy, component interface specification,
- *Technical viewpoint*: realization of logical components by software modules, run time objects, deployment and scheduling of run time objects onto the software and hardware platform.

For the design of architecture at a logical level we suggest hierarchies of logical components, on which a detailed design can be based. The difference between architecture and detailed design [13] is expressed as follows:

- Architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design.

– Design is concerned with the modularization and detailed interfaces of the design elements, their algorithms and procedures, and the data types needed to support the architecture and to satisfy the requirements.

A detailed design is mandatory for systematic module implementation and verification and later system integration, which is the key step in system development, including integration verification.

2.3 Empirical and Experimental Evaluation of Software Engineering Principles and Methods

We have listed a number of principles that are widely accepted as helpful and valid in the development of large software systems. In addition to these principles we find suggestions such as agile development (see scrum). Another example of recent claims for improvement are so-called service-oriented architectures where it is claimed that these architectures lead to a much better structuring in particular, of business and web-based systems. A third example, which is more on the programming language side is aspect-oriented programming that claims to lead to better structured programs which are easier to change and to maintain. A scientific justification or disproof of such claims is overdue.

2.3.1 Justifying Claims about Principles and Methods

One idea is that we can use empirical results and experiments to prove that certain methods and principles are effective. However, this idea runs into severe difficulties. One difficulties is that still the field of software and systems engineering is changing too fast such that experiences made a few years ago may not apply for the next system because our hardware changes, our software techniques change, the size of the system changes.

The other problem is the fuzziness of statistical analysis. There are too many factors, which influence the success of software projects. If we apply a method in one project and not in the other and then compare both projects it is hardly possible to conclude from the success of the first project the effectiveness of the method. There are so many additional project influences such that it is hard to determine which one is dominantly responsible for project success.

A typical example is the application of model-based technology. When doing so the idea is that we model the requirements and architecture. If we then conclude that there is an improvement in the effectiveness of software development by using model-based techniques, the true reason might be that it is not so much the application of modelling techniques but rather the systematic way of dealing with requirements and architecture.

Of course, we have clear indications that firm systematic development approaches improve the quality of software but, of course, we always have to ask about the costs of the development taking to account that software development is very costly. It is not possible just to look at the methods and their effects on development quality. We

have also to determine how cost-effective a method is and then relate the positive effects of the methods to its costs as addressed under keywords as “design to cost”.

3 Scientific Foundations of Engineering Methods

First of all, formalization is a general method in science. It has been created as a technique in mathematics and also in philosophical and mathematical logic with the general aim to express propositions and to argue about them in a fully objective way. In some sense it is the ultimate goal of science to deal with its themes in an absolutely objective way.

Only in the last century, formal logic has entered into engineering. First of all, logic has been turned into an engineering concept when designing switching circuits and also by the logic of software systems. Secondly, the logical approaches help in developing software and digital hardware – after all code is a formal artefact defining a logic of execution.

3.1 About the Concept of Engineering Methods

A method defines “how to do or make something”.

A method is a very general term and has a flavour that it is a way to reach a particular goal, where the steps to reach that goal are very well defined such that skilled people can perform them. Engineers therefore heavily use methods as ways to reach their sub-goals in the development process.

3.2 Why Formal Specification and Verification is Not Enough

Formal development methods that mainly aim at formal specification and verification are not sufficient for the challenge to make software systems reliable and functionally safe such that they fulfil valid requirements of their users with expected quality and are constructed in cost effective ways. Pure formalization and verification can only prove a correct relationship between formal specifications and implementations but cannot prove that the systems meet valid requirements.

Therefore the project on the verifying compiler (see [11]) has an essential weakness since it only addresses partial aspects of correctness but not validity of requirements.

3.3 Importance of the Formalization of Engineering Concepts

Engineering concepts in systems and software development are mostly complex and abstract. Therefore they are difficult to define properly, to understand and justify. We see a great potential for formalization in the precise definition of terms and notions in engineering and in the formal analysis of engineering techniques. We see a significant

discrepancy between a formal method and the scientific method of formalization and formal foundations of engineering method.

3.4 The Role of Automation and Tools

Any methods used in the engineering of software systems are only helpful if they scale up and are cost effective. This means they have to be supported to a great deal by automation through tools.

Here formal methods actually can offer something because any tool support requires a sufficient amount of formalization. The better a method can be formalized the better it can be automatized and supported by tools.

4 Seamless Model Based Development

Model Based Engineering (MBE) is a software development methodology which focuses on creating models, or abstractions, more close to particular domain concepts rather than programming, computing and algorithmic concepts. It aims at increasing productivity by maximizing compatibility between systems, simplifying the process of design, increasing automation, and promoting communication between individuals and teams working on the system.

4.1 What are Helpful Models?

A model is an *appropriate abstraction for a particular purpose*. This is of course, a very general connotation of the concept of a model. Having a closer look, models have to be represented and communicated in order to be useful in software engineering. We should keep in mind, however, that an appropriate "Gedankenmodell", which provides a particular way and abstraction of how to think about a problem is useful for the engineer even without an explicit representation of models for communication. Using Gedankenmodells means to think and argue about a system in a goal directed way in terms of useful models.

We are interested not only in individual models but also in *modelling concepts*. These are hopefully proven techniques to derive certain abstractions for specific purposes. Here is an important strength of modelling, namely that effective modelling concepts provide useful patterns of engineering such as design patterns.

4.1.1 Modelling Requirements

Capturing and documenting requirements is one of the big challenges in the evolution of software intensive systems. As well known, we have to distinguish between functional requirements and quality requirements. We concentrate in the following mainly on functional requirements. For functional requirements modelling techniques help since we can describe the functionality of software-intensive systems by using formal specification techniques.

Systematic requirements engineering produces complete formal specifications of the interface behaviour of the system under construction. Since for many systems the functionality is much too large to be captured in one monolithic specification, specifications have to be structured. For instance, techniques are needed to structure the functionality of large multifunctional systems by hierarchies of sub-functions. The system model, briefly introduced in the appendix of [9], allows specifying the interface behaviour of the sub-functions and at the same moment using modes to specify how they are dependent and to capture the feature interactions. Worked-out in full detail state-machines with input and output capture the behaviour of the sub-services describing the interactions and dependencies between the different sub-functionalities with the help of modes. Such descriptions are worked-out starting from use-cases.

Thus way we obtain a fully formalized high level functional specification of a system structured into a number of sub-functions.

4.1.2 Modelling Architecture

A key task is system evolution is the modelling of architectures. Having modelled the function hierarchy as described above a next step is to design a logical component architecture capturing the decomposition of the system into logical components, again in a hierarchical style. However, logical component architectures provide completely different views in contrast to function hierarchies derived in requirements engineering.

How are the two views related? The interface behaviour of the logical architecture hierarchy has to be correct with respect to the overall functionality as specified by the function.

4.1.3 From Requirements and Architecture to Implementation and Integration

Having worked out a complete description of requirements and architecture all further development steps are very much guided by these artefacts. First of all, the architecture model provides specifications of the components and modules. On this basis, we can do a completely independent implementation of the individual components (following the principle of separation of concerns and modularity), say, in terms of state machine models. From these state machine models we can generate code. Moreover, we can even formally verify architectures before having worked out their implementation. Architectures with components described by state machines can be tested even before implementation and from them we can generate test cases for integration tests. Provided, the architecture is described and specified in detail, we can derive and verify from the architectural specification also properties of the overall functionality as specified by the function hierarchy specification of the system.

Architecture design can be carried out rigorously formally. This is, of course, not so easy for large systems. It is a notable property of formal methods whether they scale and how they may be applied in a lightweight manner.

If early architecture verification is done accurately and if modules are verified properly then during system integration we have not to be afraid of finding many new bugs. In fact, if architecture verification and component verification are not done properly, significant bugs are discovered much too late during system integration, as it is

the case in practice today, where architectures are not verified and modules are not properly specified. Then module verification cannot be done properly; all problems of systems show up only much too late during system integration and verification.

4.2 Modelling Systems

Based on a comprehensive set of concepts for modelling systems – as shortly outlined in the appendix of [9] – an integrated system description approach can be obtained.

4.2.1 The Significance of Precise Terminology and Clean System Concepts

One of the significant advantages of formal and mathematical techniques in software and systems engineering is not just the possibility to increase automatic tool support, to formalize and to write formal specifications and to do formal verification. Perhaps, equally important is to have clear notions and precise terminology. In many areas of software and systems engineering terms are complex abstract notions, are fuzzy and not properly chosen and made precise. Simple examples are terms like “function” or “feature” or “service”, which are frequently used in software and systems engineering without a proper definition. As a result understanding between the engineers is limited and a lot of time is wasted in confusing discussions.

4.2.2 An Integrated Model for System Specification and Implementation

A specified and implemented system is described by (for models of the used formal concepts see appendix of [9]):

- an identifier k, the system name,
- an interface behaviour specification consisting of
  - a syntactic interface description *synif(k)*
  - an interface behaviour specification *specif(k)*
- an implementation design *dsgn(k)* for the interface syntactic interface, being either
  - an interpreted architecture *dsgn(k)*,
  - a state machine *dsgn(k)*.

We end up with a hierarchical system model that way, where systems are decomposed into architectures with subsystems called their *components* that again can be decomposed via architectures into subsystems until these are finally realized by state machines. We assume that all identifiers in the hierarchy are unique. Then a hierarchical system with name k defines a set of subsystems subs(k).

Each subsystem as part of a specified and implemented system then has its own specification and implementation. A system has an *implemented behaviour* by considering only the implementation designs in the hierarchy and a *specified behaviour* by the interface specifications included in the hierarchy.

A system k is called *correct*, if the interface abstraction of its implementation A = dsgn(k) has an interface abstraction F_A that is a refinement of its interface specification specif(k) = F:

$$F \approx_{ref} F_A$$

On the basis of such a formal system model we can classify faults. A system is called *fully correct*, if all its sub-systems are correct. A system is called *faulty*, if some of its subsystems are not correct. A system fault of a system implemented by some architecture is called *architecture fault*, if the interface behaviour of the specified architecture is not a refinement of the interface specification of the system. A fault is called *component fault*, if the implemented behaviour of a subsystem is not a refinement of the specified behaviour. A clear distinction between architecture faults and component faults is not possible, in practice, today due to insufficient architecture specification (see [15]).

4.2.3 From Models to Code

If, as part of the architectural description of a system, for each component an executable model in terms of state-machines is given then we have an executable system description. With such a description we can generate test cases both at the component level, at the integration level and at the system test level. Then if the code is handwritten it can be tested by the test cases generated from the system model. On the other hand it is also possible to generate the code directly from the complete system description. In this case it does not make much sense to generate test cases from the models since the generated code should exactly reflect the behaviour of the models. Only if there is some doubt whether the code generator is correct to makes sense to use test cases to certify the code generator.

4.2.4 Software product lines

It is more and more typical that software development is no longer done from scratch where a completely new software system is developed in a green field approach. It is much more typical that systems are developed in the following constellations:

- A system is to be developed to replace an existing one where parts of the older systems will be reused.
- Significant parts of a system are taken from standard implementations and standard components available.
- A system evolution is carried out where a system is step by step changed and refined in adapted to new situations.
- A software family has to be produced where typically a large number of systems, with very similar functionalities, have to be created, is such cases a common basis is to be used.
- A platform is created which allows implementing a family of systems with similarities.

Often several of these development patterns are applied side by side. Model-based technologies are very well suited to support these different types of more industrialized software development.

4.2.5 Modular System Design, Specification, and Implementation

It is essential to distinguish between

- the architectural design of a system and
- the implementation of the components specified by an architectural design.

An architectural design consists in the identification of its components, their specification and the way they interact and form the architecture.

If the architectural design and the specification of the components is precise enough then we are able to determine the result of the cooperation of the components of the architectures, according to their specification, even without providing an implementation. If the specifications are addressing behaviour of the components and if the design is modular, then the behaviour of the architecture can be derived from the behaviour of its components and the way they are connected. In other words, in this case architecture has a specified behaviour. This specified behaviour has to be put into relation with the specification of the requirements for the system.

Having this in mind, we obtain two possibilities in making use of architecture descriptions. First of all, architecture verification can be done, based on the architecture specification without having to give implementations for the components. How verification is done depends on how the components are described. If component specifications are given by abstract state machines, then the architecture can be simulated and model-checked (if it is not too big). If component specifications are given by descriptive specifications in predicate logic, then verification is possible by logical deduction. If the components are described informally only, then we can design test cases for the architecture to see whether architectures conform to system specifications.

Given interface specifications for the components we can first of all implement the components, having the specifications in mind and then verify the components with respect to their specifications. So, we have two levels of verifications, namely, *component verification* and *architecture verification*. If both verifications are done carefully enough and if the theory is modular then correctness of the system follows from both verification steps as a corollary.

Finally, if for an implemented system for a specified system and we distinguish faults in the architectural design, where we may identify in which stage which faults appear the architecture verification would fail, from faults in the component implementation. Note that only if we are careful enough with our specification techniques to be able to specify architectures independent from component implementations then the separation of component test, architecture and integration tests and system tests are meaningful.

Furthermore, for hierarchical systems the scheme of specification, design, and implementation can be iterated for each sub-hierarchy in the architecture. In any case, we may go on in an idealised top-down development as follows: We give a requirements specification for the system, we carry out an architectural design and architectural specification for the system, this delivers specifications for components and we can go on with component specifications as requirements specification for the successive step of designing and implementing the components.

4.2.6 Formal Foundation of Methods and Models

As defined, a formal method (or better a formal engineering) method applies formal techniques in engineering. Another way to make use of formalization is the justification of methods by formal theories. Examples are proofs that specification concepts are modular or that concepts of refinement are transitive or that transformation and refactoring rules are correct.

Formal justification of methods or modelling techniques is therefore important. This allows justifying methods or development rules to be used by engineers without further explicit formal reasoning.

5 Seamless Modelling

Being formal is only one attribute of a modelling technique or a development method. There are others – not less important.

5.1 Integration of Modelling Techniques

Modelling techniques and formal techniques have one thing in common. If they are applied only in isolated steps of the development process they will not show their full power and advantages well enough and, as a result, they often will not be cost effective. If just one step in the development process is formalized and formally verified and if formally verified programs are then given to a compiler, which is not verified, it is unclear whether the effect of the formal verification brings enough benefits.

The same applies to modelling. When high level models of systems are constructed and a number of results have been achieved based on these models, it is not cost effective if then in the next step the model is not used anymore and instead the work is continued by working out different models and solutions.

Tab. 1 shows a collection of modelling and development methods as well as their integration into a workflow aiming at a seamless development process by formal models and formal methods. In requirements engineering the first result should be the function hierarchy as described above. Then the logical component architectures are designed and verified by test cases generated from the specification of the functional hierarchies. For the components, test cases can be generated from the component specifications being part of the architecture. Logical test cases can be translated into test cases at the technical level. If the logical architecture is flexible enough, it is a good starting point for working out from it units of deployment, which then can be deployed and scheduled as part of the technical architecture in terms of distributed hardware structure and its operating systems.

Artifact	Based on	Formal Description	Formal method to Work Out	Validation & Verification	Generated artifacts
Business Goals		Goal trees	Logical deduction	Logical analysis	-
Requirements	System functionality and quality model	Tables with attributes and predicate logic Taxonomies	*Use cases* Formalization in predicate logics	Consistency proof Derivation of safety assertions	System assertions System test cases
Data models	Use cases	Algebraic data types E/R diagrams Class diagrams	Axiomatization	Proof of consistency and relative completeness	-
System specification	Interface model	Syntactic interface and interface assertions Abstract state machines Interaction diagrams	Stepwise refinement	Proof of safety assertions and requirements Derivation of interaction diagrams	Interaction diagrams System test cases
Architecture	Component and composition	Hierarchy of Data flow diagrams	Decomposition	Architecture verification Architecture simulation	Interaction diagrams Integration tests
Components	Component model	Syntactic interface and interface assertions Abstract state machines	Decomposition of system specification assertions	Consistency analysis	Component tests
Implementation	State machines	State transition diagrams State transition tables	Stepwise derivation of state space and state transition rules	See component verification	
Component verification	State machine runs	Proofs in predicate logics Tests	Proof of interface assertions Test case generation	-	Test runs
Integration	Interactions	Interaction diagrams	Incremental composition	-	Test runs Interaction diagrams
System verification	Interface interaction	System interface assertions System test cases	Proof of interface assertions Test case generation	-	Test runs

Tab. 1 Formal Artefacts, Models and Methods in Seamless Model Based Development

From the models of the architecture and its interface descriptions test cases for module tests as well as extensive integration test cases can be generated and executed. The same applies for system test.

5.2 Reducing Costs – Increasing Quality

One of the large potentials of formal models and techniques in development process is their effects to reduce costs. There are mainly four possibilities for cost reduction as numerated below.

1. Avoiding and finding faults early in the process,
2. Applying proven methods and techniques that are standardized and ready for use,
3. Automation of the development task and steps wherever possible,
4. Reuse of implementations, architectures requirements and development patterns wherever possible.

The last step goes into the direction of a product line engineering, which needs a highly worked out modelling and formalization approach to be able to gain all the benefits of such an approach.

6 Software Project Governance

One of the key success factors for the development and evolution of software-intensive systems is appropriate project governance. This includes all questions of organizing, managing, controlling and steering of software projects. A lot of insights have been gained over the years due to extensive experiences and learning from the many failures in software development. It is obvious that we need project governance that establishes cooperative processes and team organization, defines clear responsibilities for all the levels of the team hierarchies, that is able to deal with cost

estimation and meaningful reactions to cost overruns, understands the separation of work between different teams, to define responsibilities, to understand the competences needed and to install effective processes and techniques for planning, progress control, change management, version and configuration management as well as quality control.

Traditionally the two sides of technical and methodological software and system development and project governance are not well integrated. There is not much work relating management issues with issues of methods, techniques and formalisms. But this work is needed. Methods can only be effective if the management context is the right one for them and also by clever management project success cannot be enforced without effective development methods. We need much better insights into the interdependencies and the interplay between management techniques and development techniques.

7 Concluding Remarks: Towards a Synergy between Formal Methods and Model Based Development

Not surprisingly the synergy between formal methods and model-based development is very deep. This synergy is not exploited in enough details so far. It is certainly not sufficient for a formal development method just to provide a formalization of informal approaches like the unified modelling language UML or to develop techniques of model checking where certain models that have been worked out in the development process. A much deeper synergy is needed where appropriate formal models directly address the structure of functionality and architecture. This concept requires targeted structures of the models and their relations by refinement. Furthermore, tracing between the models must be a built-in property supporting change requests. Then the whole model structure can be continuously updated and modified in a way such that consistent system models are guaranteed in every step of development.

Putting formal methods and modelling together we end up with a powerful concept, developing formal methods and modelling for a particular purpose, addressing particular issues in the evolution of software intensive systems with the rigour of formality and its possibilities for automation and reuse. This brings in a new quality. Achieving this, however, needs a lot of research starting from useful theories, based on valid principles, finding tractable syntax, modelling the application domain and finally integrate them into the development process and supporting it with appropriate tools.

References

1. Botaschanjan, J., Broy, M., Gruler, A., Harhurin, A., Knapp, S., Kof, L., Paul, W.J., Spichkova, M.: On the correctness of upper layers of automotive systems. Formal Asp. Comput. 20(6), pp. 637-662 (2008)
2. Brooks, F.P. Jr.: The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley (1975)

3. Broy, M., Stølen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer (2001)
4. Broy, M.: A Theory of System Interaction: Components, Interfaces, and Services. In: D. Goldin, S. Smolka and P. Wegner (eds.): The New Paradigm. Springer Verlag, Berlin, pp. 41-96 (2006)
5. Broy, M., Krüger, I., Meisinger, M.: A Formal Model of Services. ACM Trans. Softw. Eng. Methodol. 16(1) (February 2007)
6. Broy, M.: The ‚Grand Challenge' in Informatics: Engineering Software-Intensive Systems. IEEE Computer, pp. 72–80, Oktober (2006)
7. Broy, M.: Model-driven architecture-centric engineering of (embedded) software intensive systems: modelling theories and architectural milestones. Innovations Syst. Softw. Eng. 3(1), pp. 75-102 (2007)
8. Broy, M.: Interaction and Realizability, In: Jan van Leeuwen, Giuseppe F. Italiona, Wiebe van der Hoek, Christoph Meinel, Harald Sack, Frantisek Plasil (eds.): SOFSEM 2007: Theory and Practice of Computer Science, Lecture Notes in Computer Science vol. 4362, pp. 29–50, Springer (2007)
9. Broy, M.: Seamless Model Driven Systems Engineering Based on Formal Models. In: Karin Breitman, Ana Cavalcanti (eds.): Formal Methods and Software Engineering. 11th International Conference on Formal Engineering Methods (ICFEM'09), Lecture Notes in Computer Science vol. 5885, pp.1-19. Springer (2009)
10. Broy, M.: Multifunctional Software Systems: Structured Modelling and Specification of Functional Requirements. Science of Computer Programming, accepted for publication
11. Hoare, T.: The verifying compiler: A grand challenge for computing research. *J. ACM* 50(1), pp. 63-69 (2003)
12. ISO DIS 26262
13. Perry, D.E., Wolf, A.L.: Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes 17(4) (October 1992)
14. Parnas, D.L.: Some Software Engineering Principles. In: Software fundamentals: collected papers by David L. Parnas, Addison-Wesley Longman Publishing Co., Inc Boston, MA, pp. 257-266 (2001)
15. Reiter, H.: Reduktion von Integrationsproblemen für Software im Automobil durch frühzeitige Erkennung und Vermeidung von Architekturfehlern. Ph. D. Thesis, Technische Universität München, Fakultät für Informatik, forthcoming

Logical Abstract Domains and Interpretations

Patrick Cousot [2,3], Radhia Cousot [3,1], and Laurent Mauborgne [3,4]

[1] Centre National de la Recherche Scientifique, Paris
[2] Courant Institute of Mathematical Sciences, New York University
[3] École Normale Supérieure, Paris
[4] Instituto Madrileño de Estudios Avanzados, Madrid

Abstract. We give semantic foundations to abstract domains consisting in first order logic formulæ in a theory, as used in verification tools or methods using SMT-solvers or theorem provers. We exhibit conditions for a sound usage of such methods with respect to multi-interpreted semantics and extend their usage to automatic invariant generation by abstract interpretation.

1 Introduction

Hoare's axiomatic logic [34,13] can be formalized as defining a program semantics $C[\![\mathtt{P}]\!]$ which is the set of inductive invariants $C[\![\mathtt{P}]\!] \triangleq \left\{ I \in A \,\middle|\, F[\![\mathtt{P}]\!](I) \sqsubseteq I \right\}$ where $F[\![\mathtt{P}]\!] \in A \to A$ is the transformer of program P in an abstract domain $\langle A, \sqsubseteq, \bot, \sqcup \rangle$, $\sqsubseteq$ is a pre-order, $\bot$ is the infimum, and the join $\sqcup$, if any, is the least upper bound (or an over-approximation) up to the pre-order equivalence. Program verification, consists in proving that a program specification $S \in A$ is implied by a program inductive invariant, that is $\exists I \in C[\![\mathtt{P}]\!] : I \sqsubseteq S$.

To be of interest, the semantics $C[\![\mathtt{P}]\!]$ must be assumed to be non-empty, which can be ensured by additional hypotheses. For example, the existence of a supremum $\top \in A$ ensures $\top \in C[\![\mathtt{P}]\!]$. A more interesting particular case is when $F[\![\mathtt{P}]\!] \in A \xrightarrow{\nearrow} A$ is increasing and $\langle A, \sqsubseteq, \bot, \sqcup \rangle$ is a cpo (or a complete lattice) in which case the $\sqsubseteq$-least fixpoint $\mathbf{lfp}^{\sqsubseteq} F[\![\mathtt{P}]\!]$ does exist, up to the pre-order equivalence, e.g. by [46], and is the strongest invariant. Besides soundness, the existence of this strongest invariant ensures the completeness of the verification method. Another interesting case is when $F[\![\mathtt{P}]\!]$ is increasing or extensive so that the iterates $F[\![P]\!]^0(\bot) \triangleq \bot$, $F[\![P]\!]^{\lambda+1}(\bot) \triangleq F[\![P]\!](F[\![P]\!]^{\lambda}(\bot))$ and $F[\![P]\!]^{\lambda}(\bot) \triangleq \bigsqcup_{\beta<\lambda} F[\![P]\!]^{\beta}(\bot)$ for limit ordinals λ (or $\forall \beta < \lambda : F[\![P]\!]^{\lambda}(\bot) \sqsupseteq F[\![P]\!]^{\beta}(\bot)$ in absence of join) do converge. The limit $F[\![P]\!]^{\epsilon}(\bot)$ is necessarily a fixpoint of $F[\![P]\!]$, which is therefore an inductive invariant, but maybe not the strongest one. When $F[\![P]\!]$ is continuous, we have $\epsilon = \omega$, the first infinite ordinal [19].

Automatic program verification can be categorized as follows.

(i) Deductive methods exploit the Floyd/Naur/Hoare proof method [13] that consists in guessing an inductive invariant $I \in A$ and proving that $F[\![\mathtt{P}]\!](I) \sqsubseteq I \wedge I \sqsubseteq S$. The end-user provides the inductive invariant $I \in A$ and a deductive system provides the correctness proof;
(ii) Fixpoint iterates generalization [12] consists in finding an over-approximation I^{λ} of the inductive definition of the iterates $F[\![P]\!]^{\lambda}(\bot)$ of $F[\![P]\!]$, machine-check that

S. Nanz (ed.), *The Future of Software Engineering*,
DOI 10.1007/978-3-642-15187-3_3,

$F[\![P]\!](I^\lambda) \sqsubseteq I^{\lambda+1}$ and $I^\lambda \sqsupseteq \bigsqcup_{\beta<\lambda} I^\beta$ for limit ordinals (proving that $\forall \lambda \geqslant 0 : F[\![P]\!]^\lambda(\bot) \sqsubseteq I^\lambda$ by recurrence and $F[\![P]\!]$ is increasing or extensive). For the limit I^ε, we have $F[\![P]\!]^\epsilon(\bot) \sqsubseteq I^\varepsilon$ so that $I^\varepsilon \sqsubseteq S$ implies $\exists I \in C[\![\mathrm{P}]\!] : I \sqsubseteq S$;

(iii) Model checking [8] considers the case when A is a finite complete lattice so that $\sqsubseteq$, $F[\![\mathrm{P}]\!]$ which is assumed to be increasing, and $\mathbf{lfp}^{\sqsubseteq} F[\![\mathrm{P}]\!]$ are all computable;

(iv) Bounded model checking [7] aims at proving that the specification is not satisfied by proving $F[\![\mathrm{P}]\!]^k(\bot) \not\sqsubseteq S$ which implies $\mathbf{lfp}^{\sqsubseteq} F[\![\mathrm{P}]\!] \not\sqsubseteq S$, e.g. by [46];

(v) Static analysis by abstract interpretation [18,20] consists in effectively computing an abstract invariant $I^\sharp$ which concretization $\gamma(I^\sharp)$ is automatically proved to satisfy $F[\![\mathrm{P}]\!](\gamma(I^\sharp)) \sqsubseteq \gamma(I^\sharp) \wedge \gamma(I^\sharp) \sqsubseteq S$. The automatic computation of $I^\sharp$ is based on (ii) in the abstract, using an abstract domain satisfying the ascending chain condition or else a widening to inductively over-approximate the concrete iterates.

By undecidability, none of these methods can be simultaneously automatic, terminating, sound, and complete on the interpreted semantics of all programs of a non-trivial programming language. Moreover all methods (i)—(iv) restrict the expressible runtime properties hence involve some form of abstraction (v).

Of particular interest is the case of program properties $A \subseteq \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ expressed as first-order formulæ on variables $\mathbb{x}$, function symbols $\mathbb{f}$, and predicates symbols $\mathbb{p}$ and $\sqsubseteq$ is logical implication $\Rightarrow$. Recent progress in SMT solvers and more generally theorem provers [4] has been exploited in deductive methods and bounded model checking on $\langle A, \Rightarrow, \mathtt{false}\rangle$ or combinations of $A_i \subseteq \mathbb{F}(\mathbb{x}, \mathbb{f}_i, \mathbb{p}_i)$, $i = 1, \ldots, n$ exploiting the Nelson-Oppen procedure [41] for combining decidable theories. We study abstract interpretation-based static analysis restricted to logical abstract domains $A \subseteq \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$, the semantic foundations, and the necessary generalization from invariant verification to invariant generation [5].

2 Terminology on First-Order Logics, Theories, Interpretations and Models

2.1 First-order logics

We define $\mathcal{B} \triangleq \{\mathit{false}, \mathit{true}\}$ to be the Booleans. The set $\mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ of first-order formulæ on a signature $\langle \mathbb{f}, \mathbb{p}\rangle$ (where $\mathbb{f}$ are the function symbols, and $\mathbb{p}$ the predicate symbols such that $\mathbb{f} \cap \mathbb{p} = \emptyset$) and variables $\mathbb{x}$, is defined as:

$\mathtt{x}, \mathtt{y}, \mathtt{z}, \ldots \in \mathbb{x}$	variables
$\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots \in \mathbb{f}^0$	constants, $\mathbb{f} \triangleq \bigcup_{n\geqslant 0} \mathbb{f}^n$
$\mathtt{f}, \mathtt{g}, \mathtt{h}, \ldots \in \mathbb{f}^n$	function symbols of arity $n \geqslant 1$
$t \in \mathbb{T}(\mathbb{x}, \mathbb{f})$ $\quad t ::= \mathtt{x} \mid \mathtt{c} \mid \mathtt{f}(t_1, \ldots, t_n)$	terms
$\mathtt{p}, \mathtt{q}, \mathtt{r}, \ldots \in \mathbb{p}^n$, $\quad \mathbb{p} \triangleq \bigcup_{n\geqslant 0} \mathbb{p}^n$	predicate symbols of arity $n \geqslant 0$, $\mathbb{p}^0 \triangleq \{\mathtt{ff}, \mathtt{tt}\}$
$a \in \mathbb{A}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ $\quad a ::= \mathtt{ff} \mid \mathtt{p}(t_1, \ldots, t_n) \mid \neg a$	atomic formulæ

[5] For example [4] is mainly on invariant verification while Ch. 12 on invariant generation by abstract interpretation is unrelated to the previous chapters using first-order logic theories.

$$
\begin{array}{lll}
e \in \mathbb{E}(\mathbb{x}, \mathbb{f}, \mathbb{p}) \triangleq \mathbb{T}(\mathbb{x}, \mathbb{f}) \cup \mathbb{A}(\mathbb{x}, \mathbb{f}, \mathbb{p}) & & \text{program expressions} \\
\varphi \in \mathbb{C}(\mathbb{x}, \mathbb{f}, \mathbb{p}) & \varphi ::= a \mid \varphi \wedge \varphi & \text{clauses in simple conjunctive normal form} \\
\Psi \in \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p}) & \Psi ::= a \mid \neg\Psi \mid \Psi \wedge \Psi \mid \exists \mathtt{x} : \Psi & \text{quantified first-order formulæ}
\end{array}
$$

In first order logics with equality, there is a distinguished predicate $=(t_1, t_2)$ which we write $t_1 = t_2$.

2.2 Theories

The set $\vec{\mathtt{x}}_\Psi$ of free variables of a formula Ψ is defined inductively as the set of variables in the formula which are not in the scope of an existential quantifier. A *sentence* of $\mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ is a formula with no free variable. A *theory* is a set of sentences [6] (called the *theorems* of the theory) and a signature, which should contain at least all the predicates and function symbols that appear in the theorems. The *language* of a theory is the set of quantified first-order formulæ that contain no predicate or function symbol outside of the signature of the theory.

The idea of theories is to restrict the possible meanings of functions and predicates in order to reason under these hypotheses. The meanings which are allowed are the meanings which make the sentences of the theory true.

2.3 Interpretations

This is better explained with the notion of interpretation of formulæ: An *interpretation* I for a signature $\langle \mathbb{f}, \mathbb{p} \rangle$ is a couple $\langle I_\mathcal{V}, I_\gamma \rangle$ such that $I_\mathcal{V}$ is a non-empty set of values, $\forall \mathtt{c} \in \mathbb{f}^0 : I_\gamma(\mathtt{c}) \in I_\mathcal{V}$, $\forall n \geqslant 1 : \forall \mathtt{f} \in \mathbb{f}^n : I_\gamma(\mathtt{f}) \in I_\mathcal{V}^n \to I_\mathcal{V}$ and $\forall n \geqslant 0 : \forall \mathtt{p} \in \mathbb{p}^n : I_\gamma(\mathtt{p}) \in I_\mathcal{V}^n \to \mathcal{B}$. Let $\mathfrak{I}$ be the class of all such interpretations I. In a given interpretation $I \in \mathfrak{I}$, an environment [6] is a function from variables to values

$$\eta \in \mathcal{R}_I \triangleq \mathbb{x} \to I_\mathcal{V} \quad \text{environments}$$

An interpretation I and an environment η satisfy a formula Ψ, written $I \models_\eta \Psi$, in the following way:

$$
\begin{array}{rclcrcl}
I \models_\eta a & \triangleq & [\![a]\!]_I \eta & \qquad & I \models_\eta \Psi \wedge \Psi' & \triangleq & (I \models_\eta \Psi) \wedge (I \models_\eta \Psi') \\
I \models_\eta \neg\Psi & \triangleq & \neg(I \models_\eta \Psi) & & I \models_\eta \exists \mathtt{x} : \Psi & \triangleq & \exists v \in I_\mathcal{V} : I \models_{\eta[\mathtt{x} \leftarrow v]} \Psi \ [7]
\end{array}
$$

where the value $[\![a]\!]_I \eta \in \mathcal{B}$ of an atomic formula $a \in \mathbb{A}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ in environment $\eta \in \mathcal{R}_I$ is

$$
\begin{array}{rcll}
[\![\mathtt{ff}]\!]_I \eta & \triangleq & \mathit{false} & \\
[\![\mathtt{p}(t_1, \ldots, t_n)]\!]_I \eta & \triangleq & I_\gamma(\mathtt{p})([\![t_1]\!]_I \eta, \ldots, [\![t_n]\!]_I \eta), & \text{where} \quad I_\gamma(\mathtt{p}) \in I_\mathcal{V}^n \to \mathcal{B} \\
[\![\neg a]\!]_I \eta & \triangleq & \neg [\![a]\!]_I \eta, & \text{where} \quad \neg \mathit{true} = \mathit{false},\ \neg \mathit{false} = \mathit{true}
\end{array}
$$

and the value $[\![t]\!]_I \eta \in I_\mathcal{V}$ of the term $t \in \mathbb{T}(\mathbb{x}, \mathbb{f})$ in environment $\eta \in \mathcal{R}_I$ is

[6] Environments are also called variable assignments, valuations, etc. For programming languages, environments may also contain the program counter, stack, etc.

[7] $\eta[\mathtt{x} \leftarrow v]$ is the assignment of v to $\mathtt{x}$ in η.

$$\begin{array}{rcll} [\![\mathtt{x}]\!]_I\eta & \triangleq & \eta(\mathtt{x}) & \\ [\![\mathtt{c}]\!]_I\eta & \triangleq & I_\gamma(\mathtt{c}), & \text{where} \quad I_\gamma(\mathtt{c}) \in I_\mathcal{V} \\ [\![\mathtt{f}(t_1,\ldots,t_n)]\!]_I\eta & \triangleq & I_\gamma(\mathtt{f})([\![t_1]\!]_I\eta,\ldots,[\![t_n]\!]_I\eta), & \text{where} \quad I_\gamma(\mathtt{f}) \in {I_\mathcal{V}}^n \to I_\mathcal{V},\ n \geqslant 1 \end{array}$$

In addition, in a first-order logic with equality the interpretation of equality is always

$$I \models_\eta t_1 = t_2 \quad \triangleq \quad [\![t_1]\!]_I\eta =_I [\![t_2]\!]_I\eta$$

where $=_I$ is the unique reflexive, symmetric, antisymmetric, and transitive relation on $I_\mathcal{V}$.

2.4 Models

An interpretation $I \in \mathfrak{I}$ is said to be a *model* of Ψ when $\exists\eta : I \models_\eta \Psi$ (i.e. I makes Ψ true). An interpretation is a *model* of a theory iff it is a model of all the theorems of the theory (i.e. makes true all theorems of the theory). The class of all models of a theory $\mathcal{T}$ is

$$\begin{array}{rl} \mathfrak{M}(\mathcal{T}) \triangleq & \{I \in \mathfrak{I} \mid \forall \Psi \in \mathcal{T} : \exists\eta : I \models_\eta \Psi\} \\ = & \{I \in \mathfrak{I} \mid \forall \Psi \in \mathcal{T} : \forall\eta : I \models_\eta \Psi\} \end{array}$$

since if Ψ is a sentence and if there is a I and a η such that $I \models_\eta \Psi$, then for all η', $I \models_{\eta'} \Psi$.

Quite often, the set of sentences of a theory is not defined extensively, but using a (generally finite) set of axioms which generate the set of theorems of the theory by implication. A theory is said to be *deductive* iff it is closed by deduction, that is all theorems that are true on all models of the theory are in the theory.

The theory of an interpretation I is the set of sentences Ψ such that I is a model of Ψ. Such a theory is trivially deductive and satisfiable (i.e. has at least one model).

2.5 Satisfiability and validity (modulo interpretations and theory)

A formula Ψ is *satisfiable* (with respect to the class of interpretations $\mathfrak{I}$ defined in Sect. **2.3**) if there exists an interpretation I and an environment η that make the formula true ($\mathsf{satisfiable}(\Psi) \triangleq \exists I \in \mathfrak{I} : \exists\eta : I \models_\eta \Psi$). A formula is *valid* if all such interpretations make the formula true ($\mathsf{valid}(\Psi) \triangleq \forall I \in \mathfrak{I} : \forall\eta : I \models_\eta \Psi$). The negations of the concepts are *unsatisfiability* ($\neg\mathsf{satisfiable}(\Psi) = \forall I \in \mathfrak{I} : \forall\eta : I \models_\eta \neg\Psi$) and *invalidity* ($\neg\mathsf{valid}(\Psi) = \exists I \in \mathfrak{I} : \exists\eta : I \models_\eta \neg\Psi$). So Ψ is satisfiable iff $\neg\Psi$ is invalid and Ψ is valid iff $\neg\Psi$ is unsatisfiable.

These notions can be put in perspective in *satisfiability and validity modulo interpretations* $\mathcal{I} \in \wp(\mathfrak{I})$, where we only consider interpretations $I \in \mathcal{I}$. So $\mathsf{satisfiable}_\mathcal{I}(\Psi) \triangleq \exists I \in \mathcal{I} : \exists\eta : I \models_\eta \Psi$ and $\mathsf{valid}_\mathcal{I}(\Psi) \triangleq \forall I \in \mathcal{I} : \forall\eta : I \models_\eta \Psi$ (also denoted $\mathcal{I} \models \Psi$).

The case $\mathcal{I} = \mathfrak{M}(\mathcal{T})$ corresponds to *satisfiability and validity modulo a theory* $\mathcal{T}$, where we only consider interpretations $I \in \mathfrak{M}(\mathcal{T})$ that are models of the theory (i.e. make true all theorems of the theory). So $\mathsf{satisfiable}_\mathcal{T}(\Psi) \triangleq \mathsf{satisfiable}_{\mathfrak{M}(\mathcal{T})}(\Psi) = \exists I \in$

$\mathfrak{M}(\mathcal{T}) : \exists \eta : I \models_\eta \Psi$ and $\mathsf{valid}_\mathcal{T}(\Psi) \triangleq \mathsf{valid}_{\mathfrak{M}(\mathcal{T})}(\Psi) = \forall I \in \mathfrak{M}(\mathcal{T}) : \forall \eta : I \models_\eta \Psi$ (also denoted $\mathcal{T} \models \Psi$).

The four concepts can be extended to theories: a theory is satisfiable [8] (valid) if one (all) of the interpretations is a (are) model(s) of the theory i.e. $\mathfrak{M}(\mathcal{T}) \neq \emptyset$ (resp. $\mathfrak{M}(\mathcal{T}) = \mathfrak{I}$), and a theory is unsatisfiable (invalid) if all (one) of the interpretations make(s) each of the theorems of the theory false.

2.6 Decidable theories

A theory $\mathcal{T}$ is *decidable* iff there is an algorithm $\mathsf{decide}_\mathcal{T} \in \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p}) \to \mathcal{B}$ that can decide in finite time if a given formula is in the theory or not.

Decidable theories provide approximations for the *satisfiability problem*: a formula Ψ is satisfiable iff there is an interpretation I and an environment η such that $I \models_\eta \Psi$ is true ($\mathsf{satisfiable}(\Psi) \triangleq \exists I \in \mathfrak{I} : \exists \eta : I \models_\eta \Psi$). So a formula Ψ with free variables $\vec{\mathbf{x}}_\Psi$ is satisfiable iff the sentence $\exists \vec{\mathbf{x}}_\Psi : \Psi$ obtained from the formula by existentially quantifying the free variables is satisfiable. So if we know that this sentence is in a satisfiable theory, then the original formula is also satisfiable and, in addition, we know that it is satisfiable for all models of that theory.

$$\mathsf{decide}_\mathcal{T}(\exists \vec{\mathbf{x}}_\Psi : \Psi) \Rightarrow \mathsf{satisfiable}_\mathcal{T}(\Psi) \qquad \text{(when } \mathcal{T} \text{ is decidable and satisfiable)}$$

Proof.

$$\begin{array}{lll}
& \mathsf{decide}_\mathcal{T}(\exists \vec{\mathbf{x}}_\Psi : \Psi) & \\
\Leftrightarrow & (\exists \vec{\mathbf{x}}_\Psi : \Psi) \in \mathcal{T} & \text{⟅def. decision procedure⟆} \\
\Rightarrow & \forall I \in \mathfrak{M}(\mathcal{T}) : \exists \eta : I \models_\eta \exists \vec{\mathbf{x}}_\Psi : \Psi & \\
& & \text{⟅def. } \mathfrak{M}(\mathcal{T}) \triangleq \{I \in \mathfrak{I} \mid \forall \Psi' \in \mathcal{T} : \exists \eta' : I \models_{\eta'} \Psi'\}\text{⟆} \\
\Leftrightarrow & \forall I \in \mathfrak{M}(\mathcal{T}) : \exists \eta : I \models_\eta \Psi & \text{⟅def. } I \models_\eta \exists \mathbf{x} : \Psi \text{ in Sect. } \mathbf{2.3}\text{⟆} \\
\Rightarrow & \exists I \in \mathfrak{M}(\mathcal{T}) : \exists \eta : I \models_\eta \Psi & \text{⟅}\mathcal{T} \text{ is satisfiable so } \mathfrak{M}(\mathcal{T}) \neq \emptyset\text{⟆} \\
\Leftrightarrow & \mathsf{satisfiable}_\mathcal{T}(\Psi) & \text{⟅def. } \mathsf{satisfiable}_\mathcal{T}(\Psi) \triangleq \exists I \in \mathfrak{M}(\mathcal{T}) : \exists \eta : I \models_\eta \Psi\text{⟆} \quad \square
\end{array}$$

So the problem of satisfiability modulo a theory $\mathcal{T}$ can be approximated by decidability in $\mathcal{T}$ in the sense that if the decision is *true* then the formula is satisfiable, otherwise we don't know in general.

The same result holds for validity:

$$\mathsf{decide}_\mathcal{T}(\forall \vec{\mathbf{x}}_\Psi : \Psi) \Rightarrow \mathsf{valid}_\mathcal{T}(\Psi) \qquad \text{(when } \mathcal{T} \text{ is decidable)}$$

Proof.

$$\begin{array}{lll}
& \mathsf{decide}_\mathcal{T}(\forall \vec{\mathbf{x}}_\Psi : \Psi) & \\
\Leftrightarrow & (\forall \vec{\mathbf{x}}_\Psi : \Psi) \in \mathcal{T} & \text{⟅def. decision procedure⟆}
\end{array}$$

[8] Model theorists often use "consistent" as a synonym for "satisfiable".

$$
\begin{aligned}
\Rightarrow\quad & \forall I \in \mathfrak{M}(\mathcal{T}) : \exists \eta : I \models_\eta \forall \vec{\mathbf{x}}_\Psi : \Psi \\
& \qquad \text{⟅def. } \mathfrak{M}(\mathcal{T}) \triangleq \{I \in \mathfrak{I} \mid \forall \Psi' \in \mathcal{T} : \exists \eta' : I \models_{\eta'} \Psi'\}\text{⟆} \\
\Leftrightarrow\quad & \forall I \in \mathfrak{M}(\mathcal{T}) : \forall \eta : I \models_\eta \forall \vec{\mathbf{x}}_\Psi : \Psi \\
& \qquad \text{⟅since } \forall \vec{\mathbf{x}}_\Psi : \Psi \text{ has no free variable so } I \models_\eta \forall \vec{\mathbf{x}}_\Psi : \Psi \text{ does not depend on } \eta\text{⟆} \\
\Leftrightarrow\quad & \forall I \in \mathfrak{M}(\mathcal{T}) : \forall \eta : I \models_\eta \Psi \qquad \text{⟅def. } I \models_\eta \forall \mathbf{x} : \Psi \text{ in Sect. } \mathbf{2.3}\text{⟆} \\
\Leftrightarrow\quad & \mathsf{valid}_\mathcal{T}(\Psi) \qquad \text{⟅}\mathsf{valid}_\mathcal{T}(\Psi) \triangleq \forall I \in \mathfrak{M}(\mathcal{T}) : \forall \eta : I \models_\eta \Psi\text{⟆} \quad \square
\end{aligned}
$$

It is possible to obtain implications in the other direction so that we solve exactly the validity or satisfiability problem, when the theory is deductive.

$$\mathsf{valid}_\mathcal{T}(\Psi) \Leftrightarrow \mathsf{decide}_\mathcal{T}(\forall \vec{\mathbf{x}}_\Psi : \Psi) \qquad \text{(when } \mathcal{T} \text{ is decidable and deductive)}$$

Proof. $\mathcal{T}$ is deductive, hence all valid sentences are theorems of the theory, so if $\mathsf{valid}_\mathcal{T}(\Psi)$ then $\forall \vec{\mathbf{x}}_\Psi : \Psi$ is a valid sentence of $\mathcal{T}$ and so it is in $\mathcal{T}$. $\square$

From that, we can obtain satisfiability of any formula:

$$\mathsf{satisfiable}_\mathcal{T}(\Psi) \Leftrightarrow \neg\,(\mathsf{decide}_\mathcal{T}(\forall \vec{\mathbf{x}}_\Psi : \neg\Psi)) \quad \text{(when } \mathcal{T} \text{ is decidable and deductive)}$$

Proof.

$$
\begin{aligned}
& \mathsf{satisfiable}_\mathcal{T}(\Psi) \\
\Leftrightarrow\quad & \neg\,(\mathsf{valid}_\mathcal{T}(\neg\Psi)) \qquad \text{⟅def. satisfiable}_\mathcal{T} \text{ and valid}_\mathcal{T} \text{ in Sect. } \mathbf{2.5}\text{⟆} \\
\Leftrightarrow\quad & \neg\,(\mathsf{decide}_\mathcal{T}(\forall \vec{\mathbf{x}}_\Psi : \neg\Psi)) \qquad \text{⟅since } \mathcal{T} \text{ is decidable and deductive⟆} \quad \square
\end{aligned}
$$

But in many tools, decision of formulæ with universal quantifiers is problematic. If we want an exact resolution of satisfiability using just existential quantifiers, we need stronger hypotheses. One sufficient condition is that the theory is *complete*. In the context of classical first order logic, a theory can be defined to be complete if for all sentences Ψ in the language of the theory, either Ψ is in the theory or $\neg\Psi$ is in the theory.

$$\mathsf{satisfiable}_\mathcal{T}(\Psi) \Leftrightarrow (\mathsf{decide}_\mathcal{T}(\exists \vec{\mathbf{x}}_\Psi : \Psi)) \qquad \text{(when } \mathcal{T} \text{ is decidable and complete)}$$

Proof. Assume $\mathcal{T}$ is complete. Then, either $\exists \vec{\mathbf{x}}_\Psi : \Psi \in \mathcal{T}$, in which case $\mathsf{decide}_\mathcal{T}(\exists \vec{\mathbf{x}}_\Psi : \Psi)$ returns *true* and we conclude $\mathsf{satisfiable}_\mathcal{T}(\Psi)$. Or $\neg(\exists \vec{\mathbf{x}}_\Psi : \Psi) \in \mathcal{T}$ so $\mathsf{decide}_\mathcal{T}(\neg(\exists \vec{\mathbf{x}}_\Psi : \Psi))$ returns *true*. But if a sentence is in the theory, that means that for all models of that theory, the sentence is true, so:

$$
\begin{aligned}
& \neg\mathsf{decide}_\mathcal{T}(\exists \vec{\mathbf{x}}_\Psi : \Psi) \\
\Leftrightarrow\quad & \mathsf{decide}_\mathcal{T}(\neg(\exists \vec{\mathbf{x}}_\Psi : \Psi)) \qquad \text{⟅}\mathcal{T} \text{ complete⟆} \\
\Leftrightarrow\quad & \neg(\exists \vec{\mathbf{x}}_\Psi : \Psi) \in \mathcal{T} \qquad \text{⟅def. decision procedure⟆} \\
\Rightarrow\quad & \forall I \in \mathfrak{M}(\mathcal{T}) : \exists \eta : I \models_\eta \neg(\exists \vec{\mathbf{x}}_\Psi : \Psi) \\
& \qquad \text{⟅def. } \mathfrak{M}(\mathcal{T}') \triangleq \{I \in \mathfrak{I} \mid \forall \Psi' \in \mathcal{T}' : \exists \eta' : I \models_{\eta'} \Psi'\}\text{⟆} \\
\Rightarrow\quad & \forall I \in \mathfrak{M}(\mathcal{T}) : \forall \eta : I \models_\eta \neg(\exists \vec{\mathbf{x}}_\Psi : \Psi) \qquad \text{⟅}\neg(\exists \vec{\mathbf{x}}_\Psi : \Psi) \text{ has no free variable⟆}
\end{aligned}
$$

$$\begin{array}{lll}
\Leftrightarrow & \forall I \in \mathfrak{M}(\mathcal{T}) : \neg(\exists \eta : I \models_\eta \exists \vec{\mathtt{x}}_\Psi : \Psi) & \wr\text{def. } \neg\wr \\
\Leftrightarrow & \forall I \in \mathfrak{M}(\mathcal{T}) : \neg(\exists \eta : I \models_\eta \Psi) & \wr\text{def. } I \models_\eta \exists \mathtt{x} : \Psi \text{ in Sect. } \mathbf{2.3}\wr \\
\Leftrightarrow & \neg(\exists I \in \mathfrak{M}(\mathcal{T}) : \exists \eta : I \models_\eta \Psi) & \wr\text{def. } \neg\wr \\
\Leftrightarrow & \neg\mathsf{satisfiable}_\mathcal{T}(\Psi) & \wr\text{def. } \mathsf{satisfiable}_\mathcal{T}(\Psi) \triangleq \exists I \in \mathfrak{M}(\mathcal{T}) : \exists \eta : I \models_\eta \Psi\wr \quad \square
\end{array}$$

It might be the case that we only need the decision procedure to be equivalent to satisfiability for a subset of the language of the theory. Then the same proof can be applied. Partial completeness can be defined in the following way: a theory is *partially complete* for a set of formulæ A iff for all $\Psi \in A$, either Ψ is in the theory or $\neg\Psi$ is in the theory.

Decision procedures will be most useful to provide approximations of implication. But in general, one needs to know if an implication is valid, and most decision procedures can only decide existential sentences. Here is the way to use decision procedures to approximate the validity of implication:

$$\begin{array}{lll}
 & \mathsf{valid}_\mathcal{T}(\forall \vec{\mathtt{x}}_\Psi \cup \vec{\mathtt{x}}_{\Psi'} : \Psi \Rightarrow \Psi') & \\
\triangleq & \forall I \in \mathfrak{M}(\mathcal{T}) : \forall \eta : I \models_\eta \forall \vec{\mathtt{x}}_\Psi \cup \vec{\mathtt{x}}_{\Psi'} : \Psi \Rightarrow \Psi' & \wr\text{def. validity modulo } \mathcal{T}\wr \\
\Leftrightarrow & \neg(\exists I \in \mathfrak{M}(\mathcal{T}) : \exists \eta : I \models_\eta \exists \vec{\mathtt{x}}_\Psi \cup \vec{\mathtt{x}}_{\Psi'} : \Psi \wedge \neg\Psi') & \wr\text{def. negation}\wr \\
\Leftrightarrow & \neg(\mathsf{satisfiable}_\mathcal{T}(\Psi \wedge \neg\Psi')) & \wr\text{def. satisfiability modulo } \mathcal{T}\wr \\
\Rightarrow & \neg\mathsf{decide}_\mathcal{T}(\exists \vec{\mathtt{x}}_{\Psi\wedge\neg\Psi'} : \Psi \wedge \neg\Psi') & \wr\text{when } \mathcal{T} \text{ is decidable and satisfiable.} \\
 & & \text{Equivalence } \Leftrightarrow \text{ holds when } \mathcal{T} \text{ is complete for } \exists \vec{\mathtt{x}}_{\Psi\wedge\neg\Psi'} : \Psi \wedge \neg\Psi'\wr
\end{array}$$

2.7 Comparison of theories

Except for decision procedures, theories are equivalent when they have the same models. A theory $\mathcal{T}_1$ is *more general* than a theory $\mathcal{T}_2$ when all models of $\mathcal{T}_2$ are models of $\mathcal{T}_1$ i.e. $\mathfrak{M}(\mathcal{T}_2) \subseteq \mathfrak{M}(\mathcal{T}_1)$. A sufficient condition for $\mathcal{T}_1$ to be more general than $\mathcal{T}_2$ is $\mathcal{T}_1 \subseteq \mathcal{T}_2$ (since $\mathcal{T}_1 \subseteq \mathcal{T}_2$ implies $\mathfrak{M}(\mathcal{T}_2) \subseteq \mathfrak{M}(\mathcal{T}_1)$). The converse holds for deductive theories. The most general theory for a given signature is the theory of $\{\mathtt{tt}\}$ (or equivalently its deductive closure), also called the theory of logical validities. If a theory $\mathcal{T}_1$ is more general than $\mathcal{T}_2$, then we have, for all formula Ψ:

$$\mathsf{satisfiable}_{\mathcal{T}_2}(\Psi) \Rightarrow \mathsf{satisfiable}_{\mathcal{T}_1}(\Psi), \quad \text{and} \quad \mathsf{valid}_{\mathcal{T}_1}(\Psi) \Rightarrow \mathsf{valid}_{\mathcal{T}_2}(\Psi)$$

A consequence is that a decision procedure for a theory can be used to approximate the satisfiability in a more general theory. Another consequence is that the implication is less often true with a more general theory.

3 Terminology on Abstract Interpretation

3.1 Interpreted concrete semantics

Abstractions in abstract interpretation [18,20], are relative to an interpreted concrete semantics $C_\mathfrak{I}[\![\mathtt{P}]\!]$ of programs $\mathtt{P}$ as defined by a program interpretation $\mathfrak{I} \in \mathfrak{J}$. That concrete semantics is defined as the set of invariants of the programs, that is post-fixpoints

in a complete lattice/cpo of concrete properties $\langle \mathcal{P}_\mathfrak{I}, \subseteq \rangle$ and a concrete transformer $F_\mathfrak{I}[\![\mathtt{P}]\!]$. We define $\mathbf{postfp}^{\leq} f \triangleq \{ x \mid f(x) \leq x \}$.

$$\begin{array}{rl}
\mathcal{R}_\mathfrak{I} & \text{concrete observables}[9] \\
\mathcal{P}_\mathfrak{I} \triangleq \wp(\mathcal{R}_\mathfrak{I}) & \text{concrete properties} \\
F_\mathfrak{I}[\![\mathtt{P}]\!] \in \mathcal{P}_\mathfrak{I} \to \mathcal{P}_\mathfrak{I} & \text{concrete transformer of program } \mathtt{P} \\
C_\mathfrak{I}[\![\mathtt{P}]\!] \triangleq \mathbf{postfp}^{\subseteq} F_\mathfrak{I}[\![\mathtt{P}]\!] \in \wp(\mathcal{P}_\mathfrak{I}) & \text{concrete semantics of program } \mathtt{P}
\end{array}$$

where the concrete transformer $F_\mathfrak{I}[\![\mathtt{P}]\!]$ of program $\mathtt{P}$ is built out of the set primitives $\emptyset$, $\mathcal{R}_\mathfrak{I}$, $\cup$, $\cap$, ... and the forward and backward transformers $\mathsf{f}, \mathsf{b} \in \mathcal{P}_\mathfrak{I} \to \mathcal{P}_\mathfrak{I}$ for assignment, the transformer $\mathsf{p} \in \mathcal{P}_\mathfrak{I} \to \mathcal{B}$ for tests, ...

Note that if the concrete transformer admits a least fixpoint, then it is enough to consider only that least fixpoint and we don't need to compute the whole set of post-fixpoints (see also Sect. **3.4**).

Example 1. In the context of invariance properties for imperative languages with program interpretation $\mathfrak{I} \in \mathfrak{J}$, we can take a concrete state to be a function from variables [10] to elements in the set $\mathfrak{I}_\mathcal{V}$, so that properties are sets of such functions.

$$\begin{array}{ll}
\mathcal{R}_\mathfrak{I} \triangleq \mathbb{X} \to \mathfrak{I}_\mathcal{V} & \text{concrete environments} \\
\mathcal{P}_\mathfrak{I} \triangleq \wp(\mathcal{R}_\mathfrak{I}) & \text{concrete invariance properties}
\end{array}$$

The transformer $F_\mathfrak{I}[\![\mathtt{P}]\!]$ for the invariance semantics is defined by structural induction on the program $\mathtt{P}$ in terms of the complete lattice operations $\langle \wp(\mathcal{R}_\mathfrak{I}), \subseteq, \emptyset, \mathcal{R}_\mathfrak{I}, \cup, \cap \rangle$ and the following local invariance transformers

$$\begin{array}{rll}
\mathsf{f}_\mathfrak{I}[\![\mathtt{x} := e]\!]P \triangleq \{\eta[\mathtt{x} \leftarrow [\![e]\!]_\mathfrak{I}\eta] \mid \eta \in P)\} & & \text{Floyd's assignment post-condition} \\
\mathsf{b}_\mathfrak{I}[\![\mathtt{x} := e]\!]P \triangleq \{\eta \mid \eta[\mathtt{x} \leftarrow [\![e]\!]_\mathfrak{I}\eta] \in P\} & & \text{Hoare's assignment pre-condition} \\
\mathsf{p}_\mathfrak{I}[\![\varphi]\!]P \triangleq \{\eta \in P \mid [\![\varphi]\!]_\mathfrak{I}\eta = \mathit{true}\} & & \text{test} \qquad \square
\end{array}$$

Example 2. The program $\mathtt{P} \triangleq \mathtt{x{=}1;\ while\ true\ \{x{=}incr(x)\}}$ with the arithmetic interpretation $\mathfrak{I}$ on integers $\mathfrak{I}_\mathcal{V} = \mathbb{Z}$ has loop invariant $\mathbf{lfp}^{\subseteq} F_\mathfrak{I}[\![\mathtt{P}]\!]$ where $F_\mathfrak{I}[\![\mathtt{P}]\!](X) \triangleq \{\eta \in \mathcal{R}_\mathfrak{I} \mid \eta(\mathtt{x}) = 1\} \cup \{\eta[\mathtt{x} \leftarrow \eta(\mathtt{x}) + 1] \mid \eta \in X\}$. The increasing chain of iterates $F_\mathfrak{I}[\![\mathtt{P}]\!]^n = \{\eta \in \mathcal{R}_\mathfrak{I} \mid 0 < \eta(\mathtt{x}) < n\}$ has limit $\mathbf{lfp}^{\subseteq} F_\mathfrak{I}[\![\mathtt{P}]\!] = \bigcup_{n \geqslant 0} F_\mathfrak{I}[\![\mathtt{P}]\!]^n = \{\eta \in \mathcal{R}_\mathfrak{I} \mid 0 < \eta(\mathtt{x})\}$. $\square$

3.2 Abstract domains

In static analysis by abstract interpretation [18,20], abstract domains are used to encapsulate abstract program properties and abstract operations (including the logical lattice structure, elementary transformers, convergence acceleration operators, etc.). An abstract domain is therefore $\langle A, \sqsubseteq, \bot, \top, \sqcup, \sqcap, \nabla, \Delta, \bar{\mathsf{f}}, \bar{\mathsf{b}}, \bar{\mathsf{p}}, \ldots \rangle$ where

[9] Examples of observables are set of states, set of partial or complete execution traces, etc.
[10] maybe including the program counter etc.

$\overline{P}, \overline{Q}, \ldots \in A$	abstract properties
$\sqsubseteq \in A \times A \to \mathcal{B}$	abstract partial order [11]
$\bot, \top \in A$	infimum, supremum
$\sqcup, \sqcap, \nabla, \Delta \in A \times A \to A$	abstract join, meet, widening, narrowing
$\ldots$	
$\bar{\mathtt{f}} \in (\mathbb{x} \times \mathbb{E}(\mathbb{x}, \mathbb{f}, \mathbb{p})) \to A \to A$	abstract forward assignment transformer
$\bar{\mathtt{b}} \in (\mathbb{x} \times \mathbb{E}(\mathbb{x}, \mathbb{f}, \mathbb{p})) \to A \to A$	abstract backward assignment transformer
$\bar{\mathtt{p}} \in \mathbb{C}(\mathbb{x}, \mathbb{f}, \mathbb{p}) \to A \to A$	abstract condition transformer

3.3 Abstract semantics

The abstract semantics of a program P is assumed to be given as a set of post-fixpoints $\overline{C}[\![\mathtt{P}]\!] \triangleq \{\overline{P} \mid \overline{F}[\![\mathtt{P}]\!](\overline{P}) \sqsubseteq \overline{P}\}$ or in least fixpoint form $\overline{C}[\![\mathtt{P}]\!] \triangleq \{\mathbf{lfp}^{\sqsubseteq}\, \overline{F}[\![\mathtt{P}]\!]\}$ (or, by the singleton isomorphism, the more frequent $\mathbf{lfp}^{\sqsubseteq}\, \overline{F}[\![\mathtt{P}]\!]$) when such a least fixpoint does exist (e.g. [46]) where $\overline{F}[\![\mathtt{P}]\!] \in A \to A$ is the abstract transformer of program P built out of the primitives $\bot, \top, \sqcup, \sqcap, \nabla, \Delta, \bar{\mathtt{f}}, \bar{\mathtt{b}}, \bar{\mathtt{p}}, \ldots$[12]. As was the case for the concrete semantics, we preferably use least fixpoints when that is possible.

3.4 Soundness of abstract domains

Soundness relates abstract properties to concrete properties using a function γ such that

$$\gamma \in A \xrightarrow{\nearrow} \mathcal{P}_{\mathfrak{I}} \quad \text{concretization [13]}$$

The soundness of abstract domains, is defined as, for all $\overline{P}, \overline{Q} \in A$,

$(\overline{P} \sqsubseteq \overline{Q}) \Rightarrow (\gamma(\overline{P}) \subseteq \gamma(\overline{Q}))$	order	$\gamma(\bot) = \emptyset$	infimum
$\gamma(\overline{P} \sqcup \overline{Q}) \supseteq (\gamma(\overline{P}) \cup \gamma(\overline{Q}))$	join	$\gamma(\top) = \top_{\mathfrak{I}}$	supremum [14]
...			

Observe that defining an abstraction consists in choosing the domain A of abstract properties and the concretization γ. So, this essentially consists in choosing a set of concrete properties $\gamma[A]$ (where $\gamma[X] \triangleq \{\gamma(x) \mid x \in X\}$) which can be exactly represented in the abstract while the other concrete properties $P \in \mathcal{P}_{\mathfrak{I}} \setminus \gamma[A]$ cannot and so must be over-approximated by some $\overline{P} \in A$ such that $P \subseteq \gamma(\overline{P})$. By assuming the existence of an element $\top$ of A with concretization $\top_{\mathfrak{I}}$, there always exists such a $\overline{P}$. For precision, the minimum one, or else the minimal ones, if any, are to be preferred.

[11] If $\sqsubseteq$ is a pre-order then A is assumed to be quotiented by the equivalence relation $\equiv \triangleq \sqsubseteq \cap \sqsubseteq^{-1}$.

[12] In general, this is more complex, with formulæ involving many fixpoints, but this simple setting already exhibits all difficulties.

[13] Given posets $\langle L, \sqsubseteq \rangle$ and $\langle P, \leqslant \rangle$, we let $L \xrightarrow{\nearrow} P$ to be the set of increasing (monotone, isotone, ...) maps of L into P.

[14] For example $\top_{\mathfrak{I}} \triangleq \mathcal{R}_{\mathfrak{I}}$ in the context of invariance properties for imperative languages.

3.5 Soundness of abstract semantics

The abstract semantics $\overline{C}[\![\mathtt{P}]\!] \in A$ is *sound* which respect to a concrete semantics $C[\![\mathtt{P}]\!]$ of a program P whenever

$$\forall \overline{P} \in A : (\exists \overline{C} \in \overline{C}[\![\mathtt{P}]\!] : \ \overline{C} \sqsubseteq \overline{P}) \Rightarrow (\exists C \in C[\![\mathtt{P}]\!] : \ C \subseteq \gamma(\overline{P}))$$

It is *complete* whenever $\forall \overline{P} \in A : (\exists C \in C[\![\mathtt{P}]\!] : C \subseteq \gamma(\overline{P})) \Rightarrow (\exists \overline{C} \in \overline{C}[\![\mathtt{P}]\!] : \overline{C} \sqsubseteq \overline{P})$. When the concrete and abstract semantics are defined in post-fixpoint form $C[\![\mathtt{P}]\!] \triangleq \mathbf{postfp}^{\subseteq} F[\![\mathtt{P}]\!]$, the soundness of the abstract semantics follows from the soundness of the abstraction in Sect. **3.4** and the soundness of the abstract transformer [18,20]

$$\forall \overline{P} \in A : F[\![\mathtt{P}]\!] \circ \gamma(\overline{P}) \subseteq \gamma \circ \overline{F}[\![\mathtt{P}]\!](\overline{P})$$

Example 3. Continuing *Ex. 1* in the context of invariance properties for imperative languages, the soundness of the abstract transformer generally follows from the following soundness conditions on local abstract transformers, for all $\overline{P} \in A$,

$$\begin{array}{rl} \gamma(\bar{\mathtt{f}}[\![\mathtt{x} := e]\!]\overline{P}) \supseteq \mathtt{f}_{\mathfrak{I}}[\![\mathtt{x} := e]\!]\gamma(\overline{P}) & \text{assignment post-condition} \\ \gamma(\bar{\mathtt{b}}[\![\mathtt{x} := e]\!]\overline{P}) \supseteq \mathtt{b}_{\mathfrak{I}}[\![\mathtt{x} := e]\!]\gamma(\overline{P}) & \text{assignment pre-condition} \\ \gamma(\bar{\mathtt{p}}[\![\varphi]\!]\overline{P}) \supseteq \mathtt{p}_{\mathfrak{I}}[\![\varphi]\!]\gamma(\overline{P}) & \text{test} \end{array} \quad \square$$

Observe that soundness is preserved by composition of increasing concretizations.

4 Abstraction of Multi-Interpreted Concrete Semantics

The interpreted concrete semantics of Sect. **3.1** is relative to one interpretation $\mathfrak{I}$ of the programming language data, functions, and predicates. But the theories used in SMT solvers can have many different models, corresponding to possible interpretations. In fact, the same holds for programs: they can be executed on different platforms, and it can be useful to collect all the possible behaviors, e.g. to provide a more general proof of correctness.

4.1 Multi-interpreted semantics

In a *multi-interpreted semantics*, we will give semantics to a program P in the context of a set of interpretations $\mathcal{I}$. Then a program property in $\mathcal{P}_{\mathcal{I}}$ provides for each interpretation in $\mathcal{I}$, a set of program observables satisfying that property in that interpretation.

$$\begin{array}{ll} \mathcal{R}_I & \text{program observables} \\ \mathcal{P}_{\mathcal{I}} \triangleq I \in \mathcal{I} \nrightarrow \wp(\mathcal{R}_I) & \text{interpreted properties} \\ \quad \simeq \wp(\{\langle I, \eta\rangle \mid I \in \mathcal{I} \wedge \eta \in \mathcal{R}_I\})\ ^{15} & \end{array}$$

The multi-interpreted semantics of a program P in the context of $\mathcal{I}$ is

[15] A partial function $f \in A \nrightarrow B$ with domain $\mathsf{dom}(f) \in \wp(A)$ is understood as the relation $\{\langle x, f(x)\rangle \in A \times B \mid x \in \mathsf{dom}(f)\}$ and maps $x \in A$ to $f(x) \in B$, written $x \in A \nrightarrow f(x) \in B$ or $x \in A \nrightarrow B_x$ when $\forall x \in A : f(x) \in B_s \subseteq B$.

$$C_{\mathcal{I}}[\![\mathtt{P}]\!] \in \wp(\mathcal{P}_{\mathcal{I}})$$

In the context of invariance properties for imperative languages with multiple program interpretations $\mathcal{I} \in \wp(\mathfrak{I})$, *Ex. 1* can be generalized by taking

$$\mathcal{R}_I \triangleq \mathbb{X} \to I_{\mathcal{V}} \quad \text{concrete interpreted environments}$$

The transformer $F_{\mathcal{I}}[\![\mathtt{P}]\!]$ for the invariance semantics is defined by structural induction on the program P in terms of the complete lattice operations $\langle \mathcal{P}_{\mathcal{I}}, \subseteq, \emptyset, \top_{\mathcal{I}}, \cup, \cap \rangle$ where $\top_{\mathcal{I}} \triangleq \{\langle I, \eta\rangle \mid I \in \mathcal{I} \wedge \eta \in \mathcal{R}_I\}$ and the following local invariance transformers

$$\begin{array}{rll}
\mathsf{f}_{\mathcal{I}}[\![\mathtt{x} := e]\!]P \triangleq \{\langle I, \eta[\mathtt{x} \leftarrow [\![e]\!]_I\eta]\rangle \mid I \in \mathcal{I} \wedge \langle I, \eta\rangle \in P)\} & & \text{assignment post-condition} \\
\mathsf{b}_{\mathcal{I}}[\![\mathtt{x} := e]\!]P \triangleq \{\langle I, \eta\rangle \mid I \in \mathcal{I} \wedge \langle I, \eta[\mathtt{x} \leftarrow [\![e]\!]_I\eta]\rangle \in P\} & & \text{assignment pre-condition} \\
\mathsf{p}_{\mathcal{I}}[\![\varphi]\!]P \triangleq \{\langle I, \eta\rangle \in P \mid I \in \mathcal{I} \wedge [\![\varphi]\!]_I\eta = \mathit{true}\} & & \text{test}
\end{array}$$

In particular for $\mathcal{I} = \{\mathfrak{I}\}$, we get the transformers of *Ex. 1*, up to the isomorphism $\iota_{\mathfrak{I}}(P) \triangleq \{\langle \mathfrak{I}, \eta\rangle \mid \eta \in P\}$ with inverse $\iota_{\mathfrak{I}}^{-1}(Q) \triangleq \{\eta \mid \langle \mathfrak{I}, \eta\rangle \in Q\}$.

The natural ordering to express abstraction (or precision) on multi-interpreted semantics is the subset ordering, which gives a lattice structure to the set of multi-interpreted properties: a property P_1 is more abstract than P_2 when $P_2 \subset P_1$, meaning that P_1 allows more behaviors for some interpretations, and maybe that it allows new interpretations. Following that ordering, we can express systematic abstractions of the multi-interpreted semantics.

4.2 Abstractions between multi-interpretations

If we can only compute properties on one interpretation $\mathfrak{I}$, as in the case of Sect. **3.1**, then we can approximate a multi-interpreted program saying that we know the possible behaviors when the interpretation is $\mathfrak{I}$ and we know nothing (so all properties are possible) for the other interpretations of the program. On the other hand, if we analyze a program that can only have one possible interpretation with a multi-interpreted property, then we are doing an abstraction in the sense that we add more behaviors and forget the actual property that should be associated with the program. So, in general, we have two sets of interpretations, one $\mathcal{I}$ is the context of interpretations for the program and the other $\mathcal{I}^\sharp$ is the set of interpretations used in the analysis. The relations between the two is a Galois connection $\langle \mathcal{P}_{\mathcal{I}}, \subseteq\rangle \underset{\alpha_{\mathcal{I}\to\mathcal{I}^\sharp}}{\overset{\gamma_{\mathcal{I}^\sharp\to\mathcal{I}}}{\leftrightarrows}} \langle \mathcal{P}_{\mathcal{I}^\sharp}, \subseteq\rangle$ where

$$\begin{array}{l}
\alpha_{\mathcal{I}\to\mathcal{I}^\sharp}(P) \triangleq P \cap \mathcal{P}_{\mathcal{I}^\sharp} \\
\gamma_{\mathcal{I}^\sharp\to\mathcal{I}}(Q) \triangleq \left\{ \langle I, \eta\rangle \;\middle|\; I \in \mathcal{I} \wedge \left(I \in \mathcal{I}^\sharp \Rightarrow \langle I, \eta\rangle \in Q\right) \right\}
\end{array}$$

Proof. Suppose $P \in \mathcal{P}_{\mathcal{I}}$ and $Q \in \mathcal{P}_{\mathcal{I}^\sharp}$. Then

$$\begin{array}{lll}
& \alpha_{\mathcal{I}\to\mathcal{I}^\sharp}(P) \subseteq Q & \\
\Leftrightarrow & P \cap \mathcal{P}_{\mathcal{I}^\sharp} \subseteq Q & \text{(def. } \alpha_{\mathcal{I}\to\mathcal{I}^\sharp}\text{)} \\
\Leftrightarrow & \forall \langle I, \eta\rangle \in P \cap \mathcal{P}_{\mathcal{I}^\sharp} : \langle I, \eta\rangle \in Q & \text{(def. } \subseteq\text{)}
\end{array}$$

$$\Leftrightarrow \quad \forall \langle I,\, \eta\rangle \in P : \langle I,\, \eta\rangle \in \mathcal{P}_{\mathcal{I}^\sharp} \Rightarrow \langle I,\, \eta\rangle \in Q \qquad \wr \text{def. } \cap \int$$

$$\Leftrightarrow \quad \forall \langle I,\, \eta\rangle \in P,\, I \in \mathcal{I} \wedge (\langle I,\, \eta\rangle \in \mathcal{P}_{\mathcal{I}^\sharp} \Rightarrow \langle I,\, \eta\rangle \in Q) \qquad \wr P \in \mathcal{P}_{\mathcal{I}} \int$$

$$\Leftrightarrow \quad P \subseteq \{\langle I,\, \eta\rangle \mid I \in \mathcal{I} \wedge (\langle I,\, \eta\rangle \in \mathcal{P}_{\mathcal{I}^\sharp} \Rightarrow \langle I,\, \eta\rangle \in Q)\} \qquad \wr \text{def. } \subseteq \int$$

$$\Leftrightarrow \quad P \subseteq \gamma_{\mathcal{I}^\sharp \to \mathcal{I}}(Q) \qquad \wr \text{def. } \gamma_{\mathcal{I}^\sharp \to \mathcal{I}} \int \quad \square$$

Note that if the intersection of $\mathcal{I}^\sharp$ and $\mathcal{I}$ is empty then the abstraction is trivially $\emptyset$ for all properties, and if $\mathcal{I} \subseteq \mathcal{I}^\sharp$ then the abstraction is the identity.

Considering the soundness of transformers defined in Sect. **3.5** for the forward assignment of Sect. **4.1**, we get, for all $P^\sharp \in \mathcal{P}_{\mathcal{I}^\sharp}$,

$$\begin{array}{lll}
 & \mathsf{f}_{\mathcal{I}}[\![\mathtt{x} := e]\!] \circ \gamma_{\mathcal{I}^\sharp \to \mathcal{I}}(P^\sharp) & \\
= & \left\{ \langle I,\, \eta[\mathtt{x} \leftarrow [\![e]\!]_I \eta]\rangle \;\middle|\; \langle I,\, \eta\rangle \in \gamma_{\mathcal{I}^\sharp \to \mathcal{I}}(P^\sharp) \right\} & \\
 & \qquad \wr \text{def. } \mathsf{f}_{\mathcal{I}}[\![\mathtt{x} := e]\!]P \triangleq \left\{ \langle I,\, \eta[\mathtt{x} \leftarrow [\![e]\!]_I \eta]\rangle \;\middle|\; \langle I,\, \eta\rangle \in P) \right\} \int & \\
= & \left\{ \langle I,\, \eta[\mathtt{x} \leftarrow [\![e]\!]_I \eta]\rangle \;\middle|\; I \in \mathcal{I} \wedge \left(I \in \mathcal{I}^\sharp \Rightarrow \langle I,\, \eta\rangle \in P^\sharp\right) \right\} & \wr \text{def. } \gamma_{\mathcal{I}^\sharp \to \mathcal{I}} \int \\
= & \left\{ \langle I,\, \eta[\mathtt{x} \leftarrow [\![e]\!]_I \eta]\rangle \;\middle|\; I \in \mathcal{I} \wedge \left(I \in \mathcal{I}^\sharp \Rightarrow (I \in \mathcal{I}^\sharp \wedge \langle I,\, \eta\rangle \in P^\sharp)\right) \right\} & \wr \text{def. } \Rightarrow \int \\
\subseteq & \left\{ \langle I,\, \eta'\rangle \;\middle|\; I \in \mathcal{I} \wedge \left(I \in \mathcal{I}^\sharp \Rightarrow \langle I,\, \eta'\rangle \in \left\{ \eta[\mathtt{x} \leftarrow [\![e]\!]_I \eta] \;\middle|\; I \in \mathcal{I}^\sharp \wedge \langle I,\, \eta\rangle \in P^\sharp \right\}\right) \right\} & \\
 & & \wr \text{def. } \subseteq \int \\
= & \left\{ \langle I,\, \eta\rangle \;\middle|\; I \in \mathcal{I} \wedge \left(I \in \mathcal{I}^\sharp \Rightarrow \langle I,\, \eta\rangle \in \mathsf{f}_{\mathcal{I}^\sharp}[\![\mathtt{x} := e]\!](P^\sharp)\right) \right\} & \\
 & \qquad \wr \text{by defining } \mathsf{f}_{\mathcal{I}^\sharp}[\![\mathtt{x} := e]\!] \in \mathcal{P}_{\mathcal{I}^\sharp} \xrightarrow{\nearrow} \mathcal{P}_{\mathcal{I}^\sharp} \text{ such that} & \\
 & \qquad\qquad \mathsf{f}_{\mathcal{I}^\sharp}[\![\mathtt{x} := e]\!]P^\sharp \triangleq \left\{ \langle I,\, \eta[\mathtt{x} \leftarrow [\![e]\!]_I \eta]\rangle \;\middle|\; I \in \mathcal{I}^\sharp \wedge \langle I,\, \eta\rangle \in P^\sharp \right\} & \\
 & \int & \\
\subseteq & \gamma_{\mathcal{I}^\sharp \to \mathcal{I}} \circ \mathsf{f}_{\mathcal{I}^\sharp}[\![\mathtt{x} := e]\!](P^\sharp) & \wr \text{def. } \gamma_{\mathcal{I}^\sharp \to \mathcal{I}} \quad . \int
\end{array}$$

Observe that $\mathsf{f}_{\mathcal{I}^\sharp}[\![\mathtt{x} := e]\!]$ and $\mathsf{f}_{\mathcal{I}}[\![\mathtt{x} := e]\!]$ have exactly the same definition. However, the corresponding post-fixpoint semantics do differ when $\mathcal{I}^\sharp \neq \mathcal{I}$ since $\langle \mathcal{P}_{\mathcal{I}^\sharp}, \subseteq\rangle \neq \langle \mathcal{P}_{\mathcal{I}}, \subseteq\rangle$.

4.3 Uniform abstraction of interpretations

In some cases, we describe the properties of the program without distinguishing the interpretations in the context of the program. This is the case for example when expressing properties that should hold for all interpretations which are possible for the program. That abstraction simply forgets the interpretations and just keeps the union of all the possible behaviors.

Example 4. That is what the Astrée analyzer [23] does when taking all possible rounding error modes for floating points computations. $\square$

The abstraction is described by $\langle \mathcal{P}_{\mathcal{I}}, \subseteq\rangle \underset{\alpha_{\mathcal{I}}}{\overset{\gamma_{\mathcal{I}}}{\leftrightarrows}} \langle \cup_{I \in \mathcal{I}} \mathcal{R}_I, \subseteq\rangle$ where

$$\gamma_{\mathcal{I}}(E) \triangleq \left\{ \langle I,\, \eta\rangle \;\middle|\; \eta \in E \right\}$$
$$\alpha_{\mathcal{I}}(P) \triangleq \left\{ \eta \;\middle|\; \exists I :\, \langle I,\, \eta\rangle \in P \right\}$$

4.4 Abstraction by a theory

Another direction for abstraction is to keep the context of interpretations and forget about the properties on variables. This is simply a projection on the first component of the pairs of interpretation and environment. In some cases it can be difficult to represent exactly an infinite set of interpretations, and we can use theories (preferably deductive with a recursively enumerable number of axioms) to represent the set of interpretations which are models of that theories. The relationship between theories and multi-interpreted semantics is expressed by the concretization function:

$$\gamma_{\mathfrak{M}}(\mathcal{T}) \triangleq \left\{ \langle I,\ \eta\rangle \;\middle|\; I \in \mathfrak{M}(\mathcal{T}) \right\}$$

Notice, though, that because the lattice of theories is not complete, there is no best abstraction of a set of interpretations by a theory in general.

Example 5. If $\mathfrak{I}$ interprets programs over the natural numbers, then by Gödel's first incompleteness theorem there is no enumerable first-order theory characterizing this interpretation, so the poset has no best abstraction of $\{\mathfrak{I}\}$. □

Once an (arbitrary) theory $\mathcal{T}$ has been chosen to abstract a set $\mathcal{I}$ of interpretations there is a best abstraction $\alpha_{\mathcal{I}\to\gamma_{\mathfrak{M}}(\mathcal{T})}(P)$ of interpreted properties in $P \in \mathcal{P}_{\mathcal{I}}$ by abstract properties in $\mathcal{P}_{\gamma_{\mathfrak{M}}(\mathcal{T})}$. However there might be no finite formula to encode this best abstraction.

5 Uninterpreted Axiomatic Semantics

We consider Hoare's axiomatic semantics [13] of a simple imperative language. The language is completely uninterpreted so programs are merely program schemata with no constraints imposed on the interpretation of functions $\mathbb{f}$ and predicates $\mathbb{p}$ [36]. Program properties are specified by formulæ in $\mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ which is a lattice $\langle \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p}), \Rrightarrow\rangle$ for the pre-order $(\Psi \Rrightarrow \Psi') \triangleq \mathsf{valid}(\Psi \Rightarrow \Psi')$ hence is considered to be quotiented by $(\Psi \Lleftarrow\Rrightarrow \Psi') \triangleq \mathsf{valid}(\Psi \Leftrightarrow \Psi')$. The axiomatic semantics $C^{\mathfrak{a}}[\![\mathtt{P}]\!]$ of a program P is assumed to be defined in post-fixpoint form [16] $C^{\mathfrak{a}}[\![\mathtt{P}]\!] \triangleq \left\{ \Psi \;\middle|\; F_{\mathfrak{a}}[\![\mathtt{P}]\!](\Psi) \Rrightarrow \Psi \right\}$ where $F_{\mathfrak{a}}[\![\mathtt{P}]\!] \in \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p}) \stackrel{\cdot}{\to} \mathbb{F}(\mathbb{x}\ \mathbb{f}, \mathbb{p})$ is the predicate transformer of program P such that $F_{\mathfrak{a}}[\![\mathtt{P}]\!](I) \Rrightarrow I$ is the verification condition for $I \in \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ to be an inductive invariant for program P. The program transformer $F_{\mathfrak{a}}$ maps programs P to a predicate transformer $F_{\mathfrak{a}}[\![\mathtt{P}]\!]$ which we assume to be defined in terms of primitive operations $\mathtt{false}, \mathtt{true}, \vee, \wedge, \mathsf{f}_{\mathfrak{a}}, \mathsf{b}_{\mathfrak{a}}, \mathsf{p}_{\mathfrak{a}}, \ldots$ such that

$\mathsf{f}_{\mathfrak{a}} \in (\mathbb{x} \times \mathbb{T}(\mathbb{x}, \mathbb{f})) \to \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p}) \to \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ $\mathsf{f}_{\mathfrak{a}}[\![\mathtt{x} := t]\!]\Psi \triangleq \exists x' : \Psi[\mathtt{x} \leftarrow x'] \wedge \mathtt{x} = t[\mathtt{x} \leftarrow x']$	axiomatic forward assignment transformer
$\mathsf{b}_{\mathfrak{a}} \in (\mathbb{x} \times \mathbb{T}(\mathbb{x}, \mathbb{f})) \to \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p}) \to \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ $\mathsf{b}_{\mathfrak{a}}[\![\mathtt{x} := t]\!]\Psi \triangleq \Psi[\mathtt{x} \leftarrow t]$	axiomatic backward assignment transformer
$\mathsf{p}_{\mathfrak{a}} \in \mathbb{C}(\mathbb{x}, \mathbb{f}, \mathbb{p}) \to \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p}) \to \mathcal{B}$ $\mathsf{p}_{\mathfrak{a}}[\![\varphi]\!]\Psi \triangleq \Psi \wedge \varphi$	axiomatic transformer for program test of condition φ

Example 6. Continuing *Ex. 2*, consider the signature with equality and one unary function, $\mathbb{f} \triangleq \mathbb{f}^0 \cup \mathbb{f}^1$ with $\mathbb{f}^0 \triangleq \{\mathtt{0}\}$, $\mathbb{f}^1 \triangleq \{\mathtt{incr}\}$ and $\mathbb{p} \triangleq \emptyset$ [16]. By Ehrenfeucht's theorem [25], this first order logic is decidable.

The program $\mathtt{P} \triangleq \mathtt{x{=}0;\ while\ true\ \{x{=}incr(x)\}}$ has loop invariant $\mathbf{lfp}^{\Rightarrow} F_{\mathfrak{a}}[\![\mathtt{P}]\!]$ where $F_{\mathfrak{a}}[\![\mathtt{P}]\!](\Psi) \triangleq (\mathtt{x} = \mathtt{0}) \vee (\exists x' : \Psi[\mathtt{x} \leftarrow x'] \wedge \mathtt{x} = \mathtt{incr(x)}[\mathtt{x} \leftarrow x']) \Leftrightarrow (\mathtt{x} = \mathtt{0}) \vee (\exists x' : \Psi[\mathtt{x} \leftarrow x'] \wedge \mathtt{x} = \mathtt{incr}(x'))$. The fixpoint iterates are

$$
\begin{aligned}
&- F_{\mathfrak{a}}[\![\mathtt{P}]\!]^0 \triangleq \mathtt{false} && \text{⟅basis⟆}\\
&- F_{\mathfrak{a}}[\![\mathtt{P}]\!]^1 \triangleq F_{\mathfrak{a}}[\![\mathtt{P}]\!](F_{\mathfrak{a}}[\![\mathtt{P}]\!]^0)\\
&= (\mathtt{x} = \mathtt{0}) \vee (\exists x' : \mathtt{false}[\mathtt{x} \leftarrow x'] \wedge \mathtt{x} = \mathtt{incr}(x'))\\
&\qquad \text{⟅def. } F_{\mathfrak{a}}[\![\mathtt{P}]\!](\Psi) \triangleq (\mathtt{x} = \mathtt{0}) \vee (\exists x' : \Psi[\mathtt{x} \leftarrow x'] \wedge \mathtt{x} = \mathtt{incr}(x'))\text{⟆}\\
&\Leftrightarrow (\mathtt{x} = \mathtt{0})\\
&- F_{\mathfrak{a}}[\![\mathtt{P}]\!]^2 \triangleq F_{\mathfrak{a}}[\![\mathtt{P}]\!](F_{\mathfrak{a}}[\![\mathtt{P}]\!]^1)\\
&= (\mathtt{x} = \mathtt{0}) \vee (\exists x_2 : (x_2 = \mathtt{0}) \wedge \mathtt{x} = \mathtt{incr}(x_2)) && \text{⟅def. } F_{\mathfrak{a}}[\![\mathtt{P}]\!]\text{⟆}\\
&\Leftrightarrow (\mathtt{x} = \mathtt{0}) \vee (\mathtt{x} = \mathtt{incr(0)}) && \text{⟅simplification⟆}\\
&- F_{\mathfrak{a}}[\![\mathtt{P}]\!]^n \triangleq \bigvee_{i=0}^{n-1} (\mathtt{x} = \mathtt{incr}^i\mathtt{(0)}) && \text{⟅induction hypothesis⟆}\\
&\qquad \text{⟅where the abbreviations are } \textstyle\bigvee \emptyset \triangleq \mathtt{false},\ \mathtt{incr}^0\mathtt{(0)} \triangleq \mathtt{0},\ \mathtt{incr}^{n+1}\mathtt{(0)} \triangleq\\
&\qquad \mathtt{incr(incr}^n\mathtt{(0))}, \text{ and } \textstyle\bigvee_{i=1}^k \varphi_i \triangleq \varphi_1 \vee \cdots \vee \varphi_k \text{ so that } F_{\mathfrak{a}}[\![\mathtt{P}]\!]^n \in \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})\text{⟆}\\
&- F_{\mathfrak{a}}[\![\mathtt{P}]\!]^{n+1} \triangleq F_{\mathfrak{a}}[\![\mathtt{P}]\!](F_{\mathfrak{a}}[\![\mathtt{P}]\!]^n) && \text{⟅recurrence⟆}\\
&= (\mathtt{x} = \mathtt{0}) \vee (\exists x' : F_{\mathfrak{a}}[\![\mathtt{P}]\!]^n[\mathtt{x} \leftarrow x'] \wedge \mathtt{x} = \mathtt{incr}(x')) && \text{⟅def. } F_{\mathfrak{a}}[\![\mathtt{P}]\!]\text{⟆}\\
&= (\mathtt{x} = \mathtt{0}) \vee (\exists x' : \Big(\bigvee_{i=0}^{n-1} (\mathtt{x} = \mathtt{incr}^i\mathtt{(0)})\Big)[\mathtt{x} \leftarrow x'] \wedge \mathtt{x} = \mathtt{incr}(x')) && \text{⟅by ind. hyp.⟆}\\
&= (\mathtt{x} = \mathtt{0}) \vee (\exists x' : \Big(\bigvee_{i=0}^{n-1} (x' = \mathtt{incr}^i\mathtt{(0)})\Big) \wedge \mathtt{x} = \mathtt{incr}(x')) && \text{⟅def. substitution⟆}\\
&\Leftrightarrow \bigvee_{i=0}^{n} (\mathtt{x} = \mathtt{incr}^i\mathtt{(0)}) \in \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p}) && \text{⟅simplification and def. } \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})\text{⟆}
\end{aligned}
$$

All iterates $F_{\mathfrak{a}}[\![\mathtt{P}]\!]^n$, $n \geqslant 0$ of $F_{\mathfrak{a}}[\![\mathtt{P}]\!]$ belong to $\mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ and form an increasing chain for $\Rightarrow$ in the poset $\langle \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p}), \Rightarrow \rangle$ up to equivalence $\Leftrightarrow\!\!\Rightarrow$. Notice above that $\bigvee_{i=1}^n \Psi_i$ is not a formula of $\mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ but a notation, only used in the mathematical reasoning, to denote a finite formula of $\mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$, precisely $\Psi_1 \vee \ldots \vee \Psi_n$. In the axiomatic semantics, the least fixpoint does not exist and the set of post-fixpoints is $\left\{ F_{\mathfrak{a}}[\![\mathtt{P}]\!]^n(\mathtt{true}) \;\middle|\; n \geq 0 \right\}$ $= \{\mathtt{true},\ \mathtt{x} = 0 \vee (\exists x' : \mathtt{x} = \mathtt{incr}(x'), \ldots\}$.

Of course the intuition would be that $\mathbf{lfp}^{\Rightarrow} F_{\mathfrak{a}}[\![\mathtt{P}]\!] = \bigvee_{i \geqslant 0} (\mathtt{x} = \mathtt{incr}^i\mathtt{(0)})$, which is an infinite formula, hence not in $\mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$. There is therefore a fundamental incompleteness in using $\mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ to express invariant of loops in programs on $\mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$.

[16] A possible interpretation would be $\mathfrak{I}_{\mathcal{V}} \triangleq \mathbb{N}$, $\gamma(\mathtt{0}) = 0$ and $\gamma(\mathtt{incr})(x) = x + 1$, another one would be with ordinals, integers, non-standard integers, or even lists where `0` is `null` and `incr(x)` returns a pointer to a node with a single field pointing to `x`.

On one hand there may be *infinitely* many exact iterates all expressible in $\mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ and on the other hand their *limit* cannot be expressed in $\mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ [10,13]. Because of this problem, refinement with predicate abstraction would typically never terminate (unless a widening is used [1]). □

6 Axiomatic Semantics Modulo Interpretations

The axiomatic semantics of Sect. **5** is completely uninterpreted that is a purely syntactic object. A link between the uninterpreted axiomatic semantics of Sect. **5** and the interpreted concrete semantics used in Sect. **3.1** or even better Sect. **4.1** must be established [10,13]. We do this in two steps, first giving universal interpretation and then considering semantics modulo a restricted set of interpretations for more precise results.

6.1 Universal interpretation of the axiomatic semantics

The universal interpretation consists in describing the properties encoded by a formula on all possible interpretations. Thus, the concretization of a formula will be given by:

$$\gamma^{\mathfrak{a}} \in \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p}) \xrightarrow{\cdot} \mathcal{P}_{\mathfrak{Z}}$$
$$\gamma^{\mathfrak{a}}(\Psi) \triangleq \{\langle I, \eta\rangle \mid I \models_{\eta} \Psi\}$$

By definition of $I \models_{\eta} \Psi$, $\gamma^{\mathfrak{a}}$ is increasing in that for all $\Psi, \Psi' \in \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$, $\Psi \Rrightarrow \Psi'$ implies that $\gamma^{\mathfrak{a}}(\Psi) \subseteq \gamma^{\mathfrak{a}}(\Psi')$.

Theorem 1. *The axiomatic semantics is sound with respect to all multi-interpreted semantics.*

Proof. To prove the soundness of the axiomatic semantics with respect to the multi-interpreted semantics in a context of interpretations $\mathcal{I}$, we first prove the soundness in the context of all possible interpretations and then use the result of Sect. **4.2** to show soundness with respect to all contexts of interpretations, which include the basic case of Sect. **3.1**.

Soundness with respect to a multi-interpreted semantics of the program can be expressed as:

$$\forall \Psi \in C^{\mathfrak{a}}[\![\mathtt{P}]\!] : \; C_{\mathfrak{Z}}[\![\mathtt{P}]\!] \subseteq \gamma^{\mathfrak{a}}(\Psi)$$

This can be proven by verifying local soundness conditions. For the forward assignment,

$$\begin{aligned}
& \gamma^{\mathfrak{a}}(\mathtt{f}_{\mathfrak{a}}[\![\mathtt{x} := t]\!]\Psi) \\
\triangleq \;& \gamma^{\mathfrak{a}}(\exists x' : \Psi[\mathtt{x} \leftarrow x'] \wedge \mathtt{x} = t[\mathtt{x} \leftarrow x']) \\
& \qquad\qquad \wr \text{def. } \mathtt{f}_{\mathfrak{a}}[\![\mathtt{x} := t]\!]\Psi \triangleq \exists x' : \Psi[\mathtt{x} \leftarrow x'] \wedge \mathtt{x} = t[\mathtt{x} \leftarrow x'] \wr \\
= \;& \{\langle I, \eta\rangle \mid I \models_{\eta} (\exists x' : \Psi[\mathtt{x} \leftarrow x'] \wedge \mathtt{x} = t[\mathtt{x} \leftarrow x'])\} \; \wr \text{def. } \gamma^{\mathfrak{a}}(\Psi) = \{\langle I, \eta\rangle \mid I \models_{\eta} \Psi\} \wr \\
= \;& \{\langle I, \eta'[\mathtt{x} \leftarrow [\![t]\!]_I \eta']\rangle \mid I \models_{\eta'} \Psi\}
\end{aligned}$$

$$\begin{array}{lll} & & \wr \text{since } I \models_\eta (\exists x' : \Psi[\mathtt{x} \leftarrow x'] \wedge \mathtt{x} = t[\mathtt{x} \leftarrow x']) \text{ if and only if } \exists \eta' : I \models_{\eta'} \Psi \\ & & \quad \text{and } \eta = \eta'[\mathtt{x} \leftarrow [\![t]\!]_\iota \eta'] \wr \\ = & \{\langle I, \eta[\mathtt{x} \leftarrow [\![t]\!]_\iota \eta]\rangle \mid \langle I, \eta\rangle \in \{\langle I, \eta\rangle \mid I \models_\eta \Psi\}\} & \wr \text{renaming } \eta' \text{ into } \eta \text{ and def. } \in \wr \\ = & \{\langle I, \eta[\mathtt{x} \leftarrow [\![t]\!]_\iota \eta]\rangle \mid \langle I, \eta\rangle \in \gamma^{\mathfrak{a}}(\Psi)\} & \wr \text{def. } \gamma^{\mathfrak{a}}(\Psi) \backsimeq \{\langle I, \eta\rangle \mid I \models_\eta \Psi\} \wr \\ = & \mathsf{f}_{\mathfrak{Z}}[\![\mathtt{x} := t]\!] \circ \gamma^{\mathfrak{a}}(\Psi) & \wr \text{def. } \mathsf{f}_{\mathfrak{Z}}[\![\mathtt{x} := t]\!]P \triangleq \{\langle I, \eta[\mathtt{x} \leftarrow [\![t]\!]_\iota \eta]\rangle \mid \langle I, \eta\rangle \in P\} \wr \end{array}$$

and similarly for backward assignment and test. □

6.2 Adding more precision: axiomatic semantics modulo interpretations

An abstraction which is sound for all possible interpretations of a program is very likely to be imprecise on the interpretations we are interested in. An easy way to be more precise is to accept as invariants all post-fixpoints for $\Rrightarrow_{\mathcal{I}}$ instead of just $\Rrightarrow$. That defines the axiomatic semantics modulo interpretations.

Definition 1. An axiomatic semantics $C^{\mathfrak{a}}_{\mathcal{I}}[\![\mathtt{P}]\!]$ modulo interpretations $\mathcal{I}$ of a program P is defined as the uninterpreted axiomatic semantics of Sect. **5** with respect to the lattice $\langle \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p}),\ \Rrightarrow_{\mathcal{I}}\rangle$ for the pre-order $(\Psi \Rrightarrow_{\mathcal{I}} \Psi') \triangleq \mathsf{valid}_{\mathcal{I}}(\Psi \Rightarrow \Psi')$ (or equivalently $\neg\mathsf{satisfiable}_{\mathcal{I}}(\Psi \wedge \neg\Psi')$) understood as quotiented by $(\Psi \Lleftarrow\Rrightarrow_{\mathcal{I}} \Psi') \triangleq \mathsf{valid}_{\mathcal{I}}(\Psi \Leftrightarrow \Psi')$. □

A remarkable point of these axiomatic semantics modulo interpretations $\mathcal{I}$ is that they all share the same program transformer $F_{\mathfrak{a}}[\![\mathtt{P}]\!]$, but indeed, if $\mathcal{I} \subseteq \mathcal{I}'$, then $C^{\mathfrak{a}}_{\mathcal{I}}[\![\mathtt{P}]\!] \supseteq C^{\mathfrak{a}}_{\mathcal{I}'}[\![\mathtt{P}]\!] \cap \mathcal{P}_{\mathcal{I}}$ meaning that semantics modulo $\mathcal{I}$ is more precise than modulo $\mathcal{I}'$.

Soundness of the axiomatics semantics modulo interpretations $\mathcal{I}$ is a consequence of the fact that it is the reduced product of the axiomatic semantics with the abstraction of the interpretations of the program. Such abstraction is trivially sound as soon as $\mathcal{I}$ contains all the interpretations in the semantics of the program we wish to consider.

6.3 Axiomatic Semantics Modulo Theory

An axiomatic semantics $C^{\mathfrak{a}}_{\mathcal{I}}[\![\mathtt{P}]\!]$ modulo interpretations $\mathcal{I}$ of Def. **1** may be definable by a theory $\mathcal{T}$ which models define $\mathcal{I}$.

Definition 2. An axiomatic semantics $C^{\mathfrak{a}}_{\mathcal{T}}[\![\mathtt{P}]\!]$ of a program P modulo a theory $\mathcal{T}$ is defined as $C^{\mathfrak{a}}_{\mathcal{T}}[\![\mathtt{P}]\!] = C^{\mathfrak{a}}_{\mathfrak{M}(\mathcal{T})}[\![\mathtt{P}]\!]$ [17]. □

Soundness again follows from the soundness of the abstraction by theory (Sect. **4.4**). There is no such best abstraction, but any sound abstraction will give sound abstraction for the axiomatic semantics modulo theory.

Since bigger sets of interpretations mean less precise semantics, more general theories will give less precise invariants.

[17] that is, by Def. **1**, as the uninterpreted axiomatic semantics of Sect. **5** with respect to the lattice $\langle \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p}),\ \Rrightarrow_{\mathcal{T}}\rangle$ for the pre-order $(\Psi \Rrightarrow_{\mathcal{T}} \Psi') \triangleq \mathsf{valid}_{\mathcal{T}}(\Psi \Rightarrow \Psi')$ (or equivalently $\neg\mathsf{satisfiable}_{\mathcal{T}}(\Psi \wedge \neg\Psi')$) understood as quotiented by $(\Psi \Lleftarrow\Rrightarrow_{\mathcal{T}} \Psi') \triangleq \mathsf{valid}_{\mathcal{T}}(\Psi \Leftrightarrow \Psi')$.

6.4 Interpreted assignment

A consequence of considering semantics modulo interpretations $\mathcal{I}$ is the ability to use simpler predicate transformers. In particular the axiomatic transformer for $\mathtt{x} := t$ is simpler if the interpretations of term t in $\mathcal{I}$ are all such that the t is invertible for variable $\mathtt{x}$. A term t is *invertible* for variable $\mathtt{x}$, with inverse $t_{\mathtt{x}}^{-1}$, in a set $\mathcal{I}$ of interpretations iff the formula $\forall \mathtt{y} : \forall \mathtt{z} : (\mathtt{y} = t[\mathtt{x} \leftarrow \mathtt{z}]) \Leftrightarrow (\mathtt{z} = t_{\mathtt{x}}^{-1}[\mathtt{x} \leftarrow \mathtt{y}])$ is true for all interpretations in $\mathcal{I}$. In this case the axiomatic assignment forward transformer can be simplified into

$$\mathtt{f}_{\mathtt{a}}[\![\mathtt{x} := t]\!]\Psi \triangleq \Psi[\mathtt{x} \leftarrow t_{\mathtt{x}}^{-1}] \qquad \text{assignment postcondition, } t \text{ invertible into } t_{\mathtt{x}}^{-1} \text{ under } \mathcal{I}$$

$$\begin{array}{lll}
 & \gamma^{\mathtt{a}}_{\mathcal{I}}(\mathtt{f}_{\mathtt{a}}[\![\mathtt{x} := t]\!]\Psi) & t \text{ invertible into } t_{\mathtt{x}}^{-1} \text{ under } \mathcal{I} \\
\triangleq & \gamma^{\mathtt{a}}_{\mathcal{I}}(\Psi[\mathtt{x} \leftarrow t_{\mathtt{x}}^{-1}]) & \wr \text{def. } \mathtt{f}_{\mathtt{a}}[\![\mathtt{x} := t]\!]\Psi \triangleq \Psi[\mathtt{x} \leftarrow t_{\mathtt{x}}^{-1}] \wr \\
= & \{\langle I, \eta\rangle \mid I \in \mathcal{I} \wedge I \models_\eta \Psi[\mathtt{x} \leftarrow t_{\mathtt{x}}^{-1}]\} & \wr \text{def. } \gamma^{\mathtt{a}}_{\mathcal{I}}(\Psi) \backsimeq \{\langle I, \eta\rangle \mid I \in \mathcal{I} \wedge I \models_\eta \Psi\} \wr \\
= & \{\langle I, \eta\rangle \mid I \in \mathcal{I} \wedge \exists x' : I \models_\eta \exists x' : \Psi[\mathtt{x} \leftarrow x'] \wedge x' = t_{\mathtt{x}}^{-1}[\mathtt{x} \leftarrow \mathtt{x}]\} & \wr \text{def. } \exists \wr \\
= & \{\langle I, \eta\rangle \mid I \in \mathcal{I} \wedge I \models_\eta (\exists x' : \Psi[\mathtt{x} \leftarrow x']) \wedge \mathtt{x} = t[\mathtt{x} \leftarrow x']\} & \\
 & & \wr t \text{ is invertible for } \mathtt{x}, \text{ with inverse } t_{\mathtt{x}}^{-1} \wr \\
= & \mathtt{f}_{\mathcal{I}}[\![\mathtt{x} := t]\!] \circ \gamma^{\mathtt{a}}_{\mathcal{I}}(\Psi) &
\end{array}$$

Observe that the uninterpreted axiomatic semantics transformer $F_{\mathtt{a}}[\![\mathtt{P}]\!]$ using the invertible assignment postcondition in any program $\mathtt{P}$ is no longer sound for all possible classes of interpretations $\mathcal{I} \in \wp(\mathfrak{I})$ but only for those for which inversion is possible.

7 Logical Abstract Domains

When performing program verification in the first-order logic setting, computing the predicate transformer is usually quite immediate. The two hard points are (1) the computation of the least fixpoint (or an approximation of it since the logical lattice is not complete) and (2) proving that the final formula implies the desired property. To solve the first case, a usual (but not entirely satisfactory) solution is to restrict the set of formulæ used to represent environments such that the ascending chain condition is enforced. For the proof of implication, a decidable theory is used. So we define logical abstract domains in the following general setting:

Definition 3. A *logical abstract domain* is a pair $\langle A, \mathcal{T}\rangle$ of a set $A \in \wp(\mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p}))$ of logical formulæ and of a theory $\mathcal{T}$ of $\mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ (which is decidable (and deductive) (and complete on A)). The abstract properties $\Psi \in A$ define the concrete properties $\gamma^{\mathtt{a}}_{\mathcal{T}}(\Psi) \triangleq \left\{ \langle I, \eta\rangle \;\middle|\; I \in \mathfrak{M}(\mathcal{T}) \wedge I \models_\eta \Psi \right\}$ relative to the models $\mathfrak{M}(\mathcal{T})$ of theory $\mathcal{T}$. The abstract order $\sqsubseteq$ on the abstract domain $\langle A, \sqsubseteq\rangle$ is defined as $(\Psi \sqsubseteq \Psi') \triangleq ((\forall \vec{\mathtt{x}}_\Psi \cup \vec{\mathtt{x}}_{\Psi'} : \Psi \Rightarrow \Psi') \in \mathcal{T})$. □

This definition of logical abstract domains is close to the logical abstract interpretation framework developped by Gulwani and Tiwari [32,31]. The main difference with our approach is that we give semantics with respect to a concrete semantics corresponding to the actual behavior of the program, whereas in the work of Gulwani and Tiwari,

the behavior of the program is assumed to be described by formulæ in the same theory as the theory of the logical abstract domain. Our approach allows the description of the abstraction mechanism, comparisons of logical abstract domains, and to provide proofs of soundness on a formal basis.

7.1 Abstraction to Logical Abstract Domains

Because $A \in \wp(\mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p}))$, we need to approximate formulæ in $\wp(\mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})) \setminus A$ by a formula in A. The alternatives [21] are either to choose a context-dependent abstraction (a different abstraction is chosen in different circumstances, which can be understood as a widening [12]) or to define an abstraction function to use a uniform context-independent approximation whenever needed. For example, the abstract assignment would be $\mathsf{f}^\sharp[\![\mathtt{x} := t]\!]\varphi \triangleq \mathsf{alpha}^{\mathcal{I}}_A(\mathsf{f}[\![\mathtt{x} := t]\!]\varphi)$. The abstraction

$$\mathsf{alpha}^{\mathcal{I}}_A \in \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p}) \to A \quad \text{abstraction (function/algorithm)}$$

abstracts a concrete first-order logic formula appearing in the axiomatic semantics into a formula in the logical abstract domain A. It is assumed to be sound in that

$$\forall \Psi \in \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p}),\ \forall I \in \mathcal{I} :\ I \models \Psi \Rightarrow \mathsf{alpha}^{\mathcal{I}}_A(\Psi) \quad \text{soundness} \tag{1}$$

It should also preferably be computable (in which case we speak of an abstraction algorithm, which can be used in the abstract semantics whereas when it is not computable it has to be eliminated e.g. by manual design of an over-approximation of the abstract operations).

Example 7 (Literal elimination). Assume that the axiomatic semantics is defined on $\mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ and that the logical abstract domain is $A = \mathbb{F}(\mathbb{x}, \mathbb{f}_A, \mathbb{p}_A)$ where $\mathbb{f}_A \subseteq \mathbb{f}$ and $\mathbb{p}_A \subseteq \mathbb{p}$. The abstraction $\mathsf{alpha}^{\mathcal{I}}_A(\Psi)$ of $\Psi \in \mathbb{F}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ can be obtained by repeating the following approximations until stabilization.

- If the formula Ψ contains one or several occurrences of a term $t \in \mathbb{f} \setminus \mathbb{f}_A$ (so is of the form $\Psi[t, \ldots, t]$), they can all be approximated by $\exists x : \Psi[x, \ldots, x]$;
- If the formula Ψ contains one or several occurrences of an atomic formula $a \in \mathbb{p} \setminus \mathbb{p}_A$ (so is of the form $\Psi[a, \ldots, a]$), this atomic formula can be replaced by `true` in the positive positions and by `false` in the negative positions.

In both cases, this implies soundness (1) and the algorithm terminates since Ψ is finite. □

Example 8 (Quantifier elimination). If the abstract domain $A \subseteq \mathbb{C}(\mathbb{x}, \mathbb{f}_A, \mathbb{p}_A)$ is quantifier-free then the quantifiers must be eliminated which is possible without loss of precision in some theories such as Presburger arithmetic (but with a potential blow-up of the formula size see e.g. [11,27,26]). Otherwise, besides simple simplifications of formulæ (e.g. replacing $\exists x : x = t \wedge \Psi[x]$ by $\Psi[t]$), a very coarse abstraction to $A \subseteq \mathbb{C}(\mathbb{x}, \mathbb{f}, \mathbb{p})$ would eliminate quantifiers bottom up, putting the formula in disjunctive normal form and eliminating the literals containing existentially quantified variables (or dually [38]),

again with a potential blow-up. Other proposals of abstraction functions (often not identified as such) include the quantifier elimination heuristics defined in Simplify [24, Sect. 5], [39, Sect. 6], or the (doubly-exponential) methods of [29,30] (which might even be made more efficient when exploiting the fact that an implication rather than an equivalence is required). □

Example 9 (Interval abstraction). Let us consider the mimimal abstraction α_m (which reduced product with the maximal abstraction, yields the interval abstraction [17,18]).

$$\begin{aligned}\alpha_m(\Psi) &\triangleq \bigwedge_{\mathtt{x}\in\mathbb{x}} \{\mathtt{c} \leqslant \mathtt{x} \mid \mathtt{c} \in \mathbb{C} \wedge \mathit{min}(\mathtt{c}, \mathtt{x}, \Psi)\}\\ \mathit{min}(\mathtt{c}, \mathtt{x}, \Psi) &\triangleq \forall \mathtt{x} : (\exists \vec{\mathtt{x}}_\Psi \setminus \{\mathtt{x}\} : \Psi) \Rightarrow (\mathtt{c} \leqslant \mathtt{x}) \wedge\\ &\qquad \forall \mathtt{m} : (\forall \mathtt{x} : (\exists \vec{\mathtt{x}}_\Psi \setminus \{\mathtt{x}\} : \Psi) \Rightarrow (\mathtt{m} \leqslant \mathtt{x})) \Rightarrow \mathtt{m} \leqslant \mathtt{c}\end{aligned}$$

Replacing the unknown constant $\mathtt{c}$ by a variable c in $\mathit{min}(\mathtt{c}, \mathtt{x}, \Psi)$, a solver might be able to determine a suitable value for $\mathtt{c}$. Otherwise the maximality requirement of $\mathtt{c}$ might be dropped to get a coarser abstraction and `true` returned in case of complete failure of the solver. □

7.2 Logical abstract transformers

For soundness with respect to a set of interpretations $\mathcal{I}$, the abstract transformers must be chosen such that [18,20]

$\mathsf{f}^\sharp \in (\mathbb{x} \times \mathbb{T}) \to A \to A$	abstract forward assignment transformer
$\forall \varphi \in A, \forall I \in \mathcal{I} :: I \models \mathsf{f}[\![\mathtt{x} := t]\!]\varphi \Rightarrow \mathsf{f}^\sharp[\![\mathtt{x} := t]\!]\varphi$	abstract postcondition
$\mathsf{b} \in (\mathbb{x} \times \mathbb{T}) \to A \to A$	abstract backward assignment transformer
$\forall \varphi \in A, \forall I \in \mathcal{I} :: I \models \mathsf{b}[\![\mathtt{x} := t]\!]\varphi \Rightarrow \mathsf{b}^\sharp[\![\mathtt{x} := t]\!]\varphi$	abstract precondition
$\mathsf{p} \in \mathbb{L} \to A \to A$	condition abstract transformer
$\forall \varphi \in A, \forall I \in \mathcal{I} :\ \mathsf{p}[\![l]\!]\varphi \Rightarrow \mathsf{p}^\sharp[\![l]\!]\varphi$	abstract test

It follows from the definition of the uninterpreted axiomatic semantics in Sect. **5** that we can define the abstract transformer to be the axiomatic transformer (under suitable closure hypothesis on A in which case they map a formula in A to a formula in A), otherwise using the abstraction α of Sect. **7.1** or using a specific local abstraction. In addition, such abstraction may be necessary on joins, as A may contain just formulæ without $\vee$.

Example 10 (Transfer function abstraction). In many static analyzers, the abstract transfer functions $\mathsf{f}^\sharp$, $\mathsf{b}^\sharp$, and $\mathsf{p}^\sharp$ of Sect. **3.2** are most often designed, proved correct and implemented by hand. This design and implementation would better be totally automatized, going back to the manual solution when automation is too inefficient or imprecise.

For logical abstract domains, the best transformers are $\mathsf{f}^\sharp[\![\mathtt{x} := t]\!] \triangleq \alpha \circ \mathsf{f}[\![\mathtt{x} := t]\!]$, $\mathsf{b}^\sharp[\![\mathtt{x} := t]\!] \triangleq \alpha \circ \mathsf{b}[\![\mathtt{x} := t]\!]$, and $\mathsf{p}^\sharp[\![\varphi]\!] \triangleq \alpha \circ \mathsf{p}[\![\varphi]\!]$ using the uninterpreted axiomatic transformers of Sect. **5**. Abstract transformers or some over-approximations (e.g. $\alpha \circ$

$\mathsf{f}[\![\mathtt{x} := t]\!] \dot{\Rightarrow} \mathsf{f}^\sharp[\![\mathtt{x} := t]\!]$) might be automatically computable using solvers (see e.g. [43] when A satisfies the ascending chain condition).

Continuing *Ex. 9*, where the abstract domain is $A = \{\bigwedge_{\mathtt{x}\in\mathbb{X}} \mathtt{c}_\mathtt{x} \leqslant \mathtt{x} \mid \forall \mathtt{x} \in \mathbb{X} : \mathtt{c}_\mathtt{x} \in \mathbb{f}^0\}$ and $\alpha = \alpha_m$, a SMT solver (e.g. with linear arithmetics or even simple inequalities [42]) might be usable when restricting Ψ in $\alpha_m(\Psi)$ to the formulæ obtained by the transformation of formulæ of A by the uninterpreted axiomatic transformers of Sect. **5**. □

Example 11 (Abstract assignment). The non-invertible assignment transformer returns a quantified formula

$$\mathsf{f}[\![\mathtt{x} := t]\!]\Psi \triangleq \exists x' : \Psi[\mathtt{x}/x'] \wedge \mathtt{x} = t[\mathtt{x}/x'] \quad \text{non-invertible assignment}$$

which may have to be abstracted to A can be done using the abstraction α of Sect. **7.1** or the widening of Sect. **7.3** or on the fly, using program specificities. For example, in straight line code outside of iteration or recursion, the existential quantifier can be eliminated

- using logical equivalence, by Skolemization where $\forall x_1 : \ldots \forall x_n : \exists y : p(x_1, \ldots, x_n, y)$ is replaced by the equi-satisfiable formula $\forall x_1 : \ldots \forall x_n : p(x_1, \ldots, x_n, f_y(x_1, \ldots, x_n))$ where f_y is a fresh symbol function;
- using a program transformation, since x' denotes the value of the variable $\mathtt{x}$ before the assignment we can use a program equivalence introducing new fresh program variable $\mathtt{x}'$ to store this value since "$\mathtt{x} := t$" is equivalent to "$\mathtt{x}' := \mathtt{x};\ \mathtt{x} := t[\mathtt{x} \leftarrow \mathtt{x}']$" [18]. We get

  $$\mathsf{f}^\sharp[\![\mathtt{x} := t]\!]\varphi \triangleq \varphi[\mathtt{x} \leftarrow \mathtt{x}'] \wedge \mathtt{x} = t[\mathtt{x} \leftarrow \mathtt{x}'] \quad \text{abstract non-invertible assignment}$$

  which may be a formula in A. This ensures soundness by program equivalence.

These local solutions cannot be used with iterations or recursions (but with a k-limiting abstraction as in bounded model checking) since a fresh auxiliary function/variable is needed for each iteration/recursive call, which number may be unbounded. □

7.3 Widening and narrowing

When the abstract domain does not satisfy the ascending chain condition, a widening is needed both to cope with the absence of infinite disjunctions and to enforce the convergence of fixpoint interation [18,12]. Designing a universal widening for logical abstract domain is difficult since powerful widenings prevent infinite evolutions in the semantic computation, evolutions which are not always well reflected as a syntactic evolution in logical abstract domains. Nevertheless, we can propose several possible widenings .

1. Widen to a finite sub-domain W of A organized in a partial order choosing $X \triangledown Y$ to be $\Psi \in W$ such that $Y \Rightarrow \Psi$ and starting from the smallest elements of W (or use a further abstraction into W as in Sect. **7.1**);

[18] This is similar to but different from Skolemization since we use auxiliary program variables instead of auxiliary functions.

2. Limit the size of formulæ to $k > 0$, eliminating new literals in the simple conjunctive normal form appearing beyond the fixed maximal size (e.g. depth) k (the above widenings are always correct but not very satisfactory, see [22]);
3. Follow the syntactic evolution of successive formulæ and reduce the evolving parts as proposed by [37] for Typed Decision Graphs.
4. Make generalizations (e.g. $l(1) \vee l(2) \vee \ldots$ implies $\exists k \geqslant 0 : l(k)$ and abstract the existential quantifier, see *Ex. 8*) or use saturation [19] [14].

8 Soundness of Unsound Abstractions

As noted in Sect. **6**, the axiomatic semantics modulo theory $\mathcal{T}$ (and thus the logical abstract domains with that semantics) are sound when all the interpretations we wish to consider for the program are models of $\mathcal{T}$. But what we see in practice is that the actual interpretations corresponding to the machine execution of programs are not models of the theories used in the program proofs. Typical examples include proofs on natural numbers, whereas the size of integers are bounded, or reasoning on floating point arithmetics as if floats behaved as reals. Indeed, it already happened that the Astrée analyzer found a buffer overrun in programs formally "proven" correct, but with respect to a theory that was an unsound approximation of the program semantics.

Still, such reasonings can give some informations about the program provided the invariant they find is precise enough. One way for them to be correct for an interpretation $\Im$ is to have one model I of the theory to agree with $\Im$ on the formulæ that appear during the computation. Formally, two theories I_1 and I_2 agree on Ψ when $\left\{\eta \,\middle|\, I_1 \models_\eta \Psi\right\} = \left\{\eta \,\middle|\, I_2 \models_\eta \Psi\right\}$

This can be achieved by monitoring the formulæ during the computation, for example insuring that the formulæ imply that numbers are always smaller than the largest machine integer. It is enough to perform this monitoring during the invariant checking phase ($F_{\mathrm{a}}[\![\mathrm{P}]\!](\Psi) \Rightarrow_{\mathcal{T}} \Psi$), so we can just check for Ψ and $F_{\mathrm{a}}[\![\mathrm{P}]\!](\Psi)$, but in some case, it can be worthwhile to detect early that the analysis cannot be correct because of an initial difference between one of the concrete interpretations and the models of the theory used to reason about the program.

9 Conclusion

The idea of using a universal representation of program properties by first-order formulæ originates from mathematical logics and underlies all deductive methods since [28,34,40] and its first automation [35]. Similarly, BDDs [5] are the universal representation used in symbolic model-checking [8].

In contrast, the success and difficulty of abstract interpretation relies on leaving open the computer representation of program properties by reasoning only in terms of abstract domains that is their algebraic structures. Data representations and algorithms have to be designed to implement the abstract domains, which is flexible and allows

[19] Saturation means to compute the closure of a given set of formulas under a given set of inference rules.

for specific efficient implementations. The alternative of using a universal representation and abstraction of the program collecting semantics for static analysis can also be considered [15]. It was intensively and successfully exploited e.g. in TVLA [45] or by abstract compilation to functional languages [3] or Prolog [9] but with difficulties to scale up.

One advantage of logical abstract domains with a uniform representation of program properties as first-order logic formulæ is the handy and understandable interface to interact with the end-user (at least when the formulæ are small enough to remain readable). It is quite easy for the end-user to help the analysis e.g. by providing assertions, invariants, hypotheses, etc. In contrast, the end-user has no access to the complex internal representation of program properties in algebraic abstract domains and so has no way to express internal properties used during the analysis, except when the abstract domain explicitly provides such an interface, e.g. for trace partitioning [44], quaternions [2], etc.

One disadvantage of the uniform representation of program properties in logical abstract domains is that it can be inefficient in particular because the growth of formulæ may be difficult to control (and may require a widening). In particular, first order-logic is the source of incompleteness at the level of the collecting semantics. The relative completeness result [10] assumes expressiveness, that is, the algebraic fixpoint semantics can be expressed in the first-order logic, which is rarely the case. Using an incomplete basis for abstraction means that some problems cannot be solved by any abstraction. Indeed, we have shown that logical abstract domains are an abstraction of the multi-interpreted semantics, and to prove more properties on that semantics, we could use second order logics [33], which is then complete but not decidable, or an ad-hoc collection of algebraic abstract domains.

The best choice, though, would be a combination of both algebraic and logical abstract domains, so that we get the best of both worlds. A possible way to investigate in that direction could be the combination of the Nelson-Oppen procedure for combining theories [41] on one hand and the reduced product on the other hand [20].

References

1. Ball, T., Podelski, A., Rajamani, S.: Relative completeness of abstraction refinement for software model checking. In: Katoen, J.P., Stevens, P. (eds.) Proc. 8^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002). pp. 158—172. LNCS 2280, Springer, Heidelberg, Grenoble (April 8–12, 2002)
2. Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static analysis and verification of aerospace software by abstract interpretation. In: AIAA Infotech@Aerospace 2010, Atlanta, Georgia. pp. AIAA 2010–3385. AIAA (20–22 April 2010)
3. Boucher, D., Feeley, M.: Abstract compilation: a new implementation paradigm for static analysis. In: Gyimothy, T. (ed.) Proc. 6th Int. Conf. on Compiler Construction, CC '96. pp. 192–207. Linköping, Lecture Notes in Computer Science 1060, Springer, Heidelberg (April 24–26, 1996)
4. Bradley, A., Manna, Z.: The Calculus of Computation, Decision procedures with Applications to Verification. Springer, Heidelberg (2007)
5. Bryant, R.E.: Graph based algorithms for boolean function manipulation. IEEE Transactions on Computers C-35, 677–691 (August 1986)

6. Chang, C., Keisler, H.: Model theory. Studies in logic and the foundation of mathematics, vol. 73. Elsevier Science, New York (1990)
7. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design 19(1), 7–34 (2001)
8. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
9. Codish, M., Søndergaard, H.: Meta-circular abstract interpretation in Prolog. In: Mogensen, T., Schmidt, D., Sudburough, I. (eds.) The Essence of Computation: Complexity, Analysis, Transformation. LNCS, vol. 2566, pp. 109–134. Springer-Verlag (2002)
10. Cook, S.: Soundness and completeness of an axiom system for program verification. SIAM J. Comput. 7, 70–80 (1978)
11. Cooper, D.: Theorem proving in arithmetic without multiplication. Machine Intelligence 91(7), 91–99 (1972)
12. Cousot, P.: Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French). Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble (21 March, 1978)
13. Cousot, P.: Methods and logics for proving programs. In: van Leeuwen, J. (ed.) Formal Models and Semantics, Handbook of Theoretical Computer Science, vol. B, chap. 15, pp. 843–993. Elsevier Science Publishers B.V., Amsterdam (1990)
14. Cousot, P.: Verification by abstract interpretation. In: Dershowitz, N. (ed.) Proc. 23rd Int. Col., ICALP '96, pp. 1–3. LNCS 1099, Springer, Heidelberg, Paderborn, Germany (July 8–12, 1996)
15. Cousot, P.: Abstract interpretation based static analysis parameterized by semantics. In: Van Hentenryck, P. (ed.) Pro. 4th Int. Symp. on Static Analysis, SAS '97, pp. 388–394. Paris, LNCS 1302, Springer, Heidelberg (September 8–10, 1997)
16. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theoretical Computer Science 277(1—2), 47–103 (2002)
17. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proc. 2^{nd} Int. Symp. on Programming. pp. 106–130. Dunod, Paris, Paris (1976)
18. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4^{th} POPL. pp. 238–252. ACM Press, Los Angeles (1977)
19. Cousot, P., Cousot, R.: Constructive versions of Tarski's fixed point theorems. Pacific Journal of Mathematics 82(1), 43–57 (1979)
20. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: 6^{th} POPL. pp. 269–282. ACM Press, San Antonio (1979)
21. Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation 2(4), 511–547 (August 1992)
22. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) Proc. 4^{th} Int. Symp. Programming Language Implementation and Logic Programming, PLILP '92. pp. 269–295. Leuven, 26–28 August 1992, LNCS 631, Springer, Heidelberg (1992)
23. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The Astrée analyser. In: Sagiv, M. (ed.) Proc. 14^{th} European Symp. on Programming Languages and Systems, ESOP '2005, Edinburg, pp. 21–30. LNCS 3444, Springer, Heidelberg (April 2–10, 2005)
24. Detlefs, D., Nelson, G., Saxe, J.: Simplify: a theorem prover for program checking. Journal of the ACM (JACM) 52(3), 365–473 (2005)
25. Ehrenfeucht, A.: Decidability of the theory of one function. Notices Amer. Math. Soc. 6, 268 (1959)

26. Ferrante, J., Geiser, J.: An efficient decision procedure for the theory of rational order. Theoretical Computer Science 4(2), 227–233 (1977)
27. Ferrante, J., Rackoff, C.: A decision procedure for the first order theory of real addition with order. SIAM Journal of Computation 4(1), 69–76 (1975)
28. Floyd, R.: Assigning meaning to programs. In: Schwartz, J. (ed.) Pro. Symp. in Applied Mathematics, vol. 19, pp. 19–32. American Mathematical Society, Providence (1967)
29. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. Conf. on Automated Deduction, CADE 21 4603 of LNAI, 167–182 (2007)
30. Ge, Y., de Moura, L.: Complete instantiation of quantified formulas in satisfiability modulo theories. Computer Aided Verification, CAV'2009 5643 of LNCS, 306–320 (2009)
31. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: 35^{th} POPL. pp. 235–246. ACM Press, San Francisco (2008)
32. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: Schwartzbach, M., Ball, T. (eds.) PLDI 2006. pp. 376–386. ACM Press, Ottawa, Ontario, Canada (11–14 June 2006)
33. Hitchcock, P., Park, D.: Induction rules and termination proofs. In: Nivat, M. (ed.) Proc. 1st Int. Colloq. on Automata, Languages and Programming. pp. 225–251. North-Holland (1973)
34. Hoare, C.: An axiomatic basis for computer programming. Communications of the Association for Computing Machinery 12(10), 576–580 (oct 1969)
35. King, J.: A Program Verifier. Ph.D. thesis, Carnegie-Mellon University (1969)
36. Luckham, D., Park, D., Paterson, M.: On formalized computer programs. J. of Computer and System Science 4(3), 220–240 (June 1970)
37. Mauborgne, L.: Abstract interpretation using typed decision graphs. Science of Computer Programming 31(1), 91–112 (May 1998)
38. McMillan, K.: Applying SAT methods in unbounded symbolic model checking. In: Brinksma, E., Larsen, K. (eds.) Computer Aided Verification, CAV'2002. vol. 2404 of LNCS, pp. 250–264 (2002)
39. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: From refutation to verification. In: Voronkov, A. (ed.) Proc. 15th Computer-Aided Verification conf. (CAV'03). LNCS, vol. 2725, pp. 14–26. Springer, Heidelberg (2003)
40. Naur, P.: Proofs of algorithms by general snapshots. BIT 6, 310–316 (1966)
41. Nelson, G., Oppen, D.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems 1(2), 245–257 (oct 1979)
42. Pratt, V.: Two easy theories whose combination is hard. Tech. rep., MIT (september 1, 1977), `boole.stanford.edu/pub/sefnp.pdf`
43. Reps, T., Sagiv, S., Yorsh, G.: Symbolic implementation of the best transformer. In: Steffen, B., Levi, G. (eds.) Proc. 5^{th} Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI 2004). pp. 252–266. LNCS 2937, Springer, Heidelberg, Venice (January 11–13, 2004)
44. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Transactions on Programming Languages and Systems 29(5) (August 2007)
45. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: 26^{th} POPL. pp. 238–252. ACM Press, an Antonio (1999)
46. Tarski, A.: A lattice theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics 5, 285–310 (1955)

Design Patterns – Past, Present & Future

Erich Gamma

IBM Rational Zurich Research Lab, Switzerland
erich_gamma@ch.ibm.com

Abstract. Design Patterns are now a 15 year old thought experiment. And today, for many, Design Patterns have become part of the standard development lexicon. This talk looks back to the origin of Design Patterns and how they evolved since their initial description. I will then show patterns in action in the context of the Eclipse and Jazz platforms. Finally, I will discuss how the Design Patterns from the book can be refactored towards a Design Pattern 2.0 version.

S. Nanz (ed.), *The Future of Software Engineering*,
DOI 10.1007/978-3-642-15187-3_4,

Evidential Authorization*

Andreas Blass,[1] Yuri Gurevich,[2] Michał Moskal[2] and Itay Neeman[3]

[1] Mathematics, University of Michigan, Ann Arbor, MI, USA
[2] Microsoft Research, Redmond, WA, USA
[3] Mathematics, UCLA, Los Angeles, CA, USA

To Bertrand Meyer, the Eiffel tower of program correctness.

Most fascinating is a feature that would make any journalist tremble. Tuyuca requires verb-endings on statements to show how the speaker knows something. Diga ape-wi means that the boy played soccer (I know because I saw him), while diga ape-hiyi means the boy played soccer (I assume). English can provide such information, but for Tuyuca that is an obligatory ending on the verb. Evidential languages force speakers to think hard about how they learned what they say they know.

—*The Economist, January 1, 2010*

Abstract. Consider interaction of principals where each principal has its own policy and different principals may not trust each other. In one scenario the principals could be pharmaceutical companies, hospitals, biomedical labs and health related government institutions. In another scenario principals could be navy fleets of different and not necessarily friendly nations. In spite of the complexity of interaction, one may want to ensure that certain properties remain invariant. For example, in the navy scenario, each fleet should have enough information from other fleets to avoid unfortunate incidents. Furthermore, one wants to use automated provers to prove invariance. A natural approach to this and many other important problems is to provide a high-level logic-based language for the principals to communicate. We do just that. Three years ago two of us presented the first incarnation of Distributed Knowledge Authorization Language (DKAL). Here we present a new and much different incarnation of DKAL that we call Evidential DKAL. Statements communicated in Evidential DKAL are supposed to be accompanied with sufficient justifications. In general, we construe the term "authorization" in the acronym "DKAL" rather liberally; DKAL is essentially a general policy language. There is a wide spectrum of potential applications of DKAL. One ambitious goal is to provide a framework for establishing and maintaining invariants.

Keywords: access control, authorization, distributed knowledge, evidential, justification, logic-level communication

* Blass was partially supported by NSF grant DMS-0653696. Part of the work reported here was performed during visits of Blass and Neeman to Microsoft.

S. Nanz (ed.), *The Future of Software Engineering*,
DOI 10.1007/978-3-642-15187-3_5,

1 Introduction

Logic-based authorization made its debut in the early 1990's with the Speaks-For calculus [17,1]. The first paper on Distributed-Knowledge Authorization Language (DKAL) was published in 2008 [12]. In the interim, substantial progress had been achieved in logic-based authorization. For brevity, let us just refer to the article [5] on SecPAL (Security Policy Authorization Language) and an informative survey of related work there.

DKAL started as an attempt to extend SecPAL but then the two languages diverged. The main motivation of DKAL was reflected in its title: distributed knowledge. The challenge was how to reason about communicating principals, each computing his own knowledge. The most conspicuous innovation of DKAL vis-à-vis SecPAL was the introduction of logic-level targeted communication among the principals involved in the given scenario. Each principal computes his[4] own knowledge, acts on it, and communicates directly with other principals. No intermediaries are necessary. Autonomous communicating principals had been considered in the literature, in particular in [3] and [8]. Yet, as far as we know, DKAL is the only language in the literature with specific logic-level rules for sending and accepting communications. Many logic-based authorization languages presume the existence of a central engine where all the reasoning occurs. For example, in the world of SecPAL, a central engine controls the resources and serves as the only knowledge manager. The only communications in the language of SecPAL are assertions of the form (P `says` φ), meaning that P says φ to the central engine. In the SecPAL implementation, different central engines do exchange information but at a lower level that is not covered by the language.

Is it important to exchange information among knowledge managers at the logic level? The answer to this question depends on what your goals are. Logic-level communication facilitates reasoning about decentralized systems (see a little example in [15]) and enables automation of such reasoning. It allows you to address new challenges. Imagine a collective of communicating principals in a paranoid world where trust is in short supply. Each principal computes his own knowledge, and not only his data but also his policy is subject to change. And yet some invariant needs to be maintained. For example, there might be just two principals. One of them is a government agency that fiddles with its policy, and the other is a company with continually evolving procedures. The invariant is that the company complies with the agency's requirements. Another example is an e-health ecosystem, and among the invariants are preservation of the safety and the privacy of the patients. Note that logic-level communication may be used to alter authorization policies in a controlled way; a step in that direction was done in [14]. In particular, this enables policy bootstrapping: One principal

[4] Of course, principals do not have to be people. They may be computers, governments, government agencies, companies, universities, etc. But it is convenient to use anthropomorphic terminology.

creates another principal and supplies the newborn with some initial policy, and then the newborn pulls necessary policy extensions from various quarters.

One interesting outcome of logic-level communication is the reemergence of quotations. Quotations were used in Speaks-For but not in recent pre-DKAL logic-based authorization languages. In particular, they are absent in SecPAL. There, communications go to the central engine and are cryptographically signed. Thus, nested quotations are superfluous. What is the point of my telling the central engine that you said φ? It knows. You said φ to it. But in the world where everybody computes his own knowledge, nested quotations are ubiquitous. For example, imagine a committee that conducted a secret vote with different members of the committee residing in different locations. The secretary confirms that two members of the committee said Yes. In a similar vein, targeted communication between principals makes little sense in the world of SecPAL.

In truly decentralized situations, a principal who wants his communications to be believed may need to provide evidence along with the communications. Such evidence may consist merely of signed statements from others but may also involve deductions. This is especially important when the recipient, say a smart card or computer server, needs to be virtually stateless (as far as the interaction in question is concerned) and therefore unable to do much fact checking but can verify justifications that are sent to it. In this paper, we present a version of DKAL, Evidential DKAL, that is geared toward applications where justifications are necessary[5].

We hasten to add that not all applications are of this sort. Imagine that you work for the 911 emergency service. You would not demand that accident reports be cryptographically signed. But you might have rules for assessing the credibility of such reports, even anonymous ones. In general, there are various kinds of applications and ways to assess evidence. In some cases you may rely on a rating system of sorts; in other cases you may decide on the basis of preponderance of evidence. Even gossip may sometimes be informative. For simplicity we treat in this paper only the case where all communications come with justifications; we postpone the general case to future work.

Another innovation of DKAL concerned authorization logic. The Speaks-For calculus is a logic of its own. Later logic-based authorization languages built on Datalog. Datalog achieves its computational efficiency by forbidding the use of functions (it is a purely relational logic apart from constraints formulated in some efficiently decidable theory) and by imposing safety requirements. In essence, the efficiency of Datalog (with or without constraints) results from its small world, consisting of just the constants that appear explicitly. In [12], the underlying logic is an extension of Datalog in which functions are allowed. So the world is no longer small, but restrictions were imposed on the ranges of some variables (so-called regular variables), and then difficult proofs showed that the efficiency is not lost. Subsequently, [14] incorporated the discovery of a strong connection with traditional constructive logic and modified the formulation of DKAL to make the connection conspicuous. In addition it showed that a useful

[5] For evidentiality in natural languages, see [2].

fragment of DKAL logic is decidable in linear time. This modification eventually allowed us to separate concerns and consider logic separately from the rest of the system. Here we treat the underlying logic and additional application-specific assumptions as a parameter.

The present paper is organized as follows. Section 2 introduces an example that will be used throughout the paper. Section 3 is an informal presentation of some of the central ideas of DKAL, which will be developed formally in subsequent sections. The world of DKAL consists of communicating agents. In accordance with the access control tradition, the agents are called principals. The principals perform various computations. Each principal has its own computation state that, in general, evolves in time[6]. Conceptually, the state splits into two parts: substrate and infostrate. Section 4 is devoted to the substrate, and Section 6 is devoted to the infostrate. Between these two, Section 5 is about the logic of DKAL. Section 7 formalizes the policies for the example of Section 2. The final Section 8 is about future work.

Acknowledgment

We are indebted to Roy DSouza, a partner architect at Microsoft, for numerous helpful conversations and for pioneering the use of DKAL in cloud federation scenarios, in particular for compliance. Roy's projects gave the impetus for the development of evidential DKAL.

2 Clinical Trials Scenario, Informal Description

As a running example in this paper, we shall use a partial description of a clinical trial of a new drug. In Section 7, the relevant policies will be written out systematically. In the present section, we describe the background situation and the policies of the parties under consideration.

2.1 Background

A pharmaceutical company developed a certain drug and tested it internally. Before bringing the drug to market, however, it is necessary to conduct a clinical trial, an external study of the safety and efficacy of the drug. Clinical trials are overseen by government regulatory bodies, such as the U.S. Food and Drug Administration (FDA), which eventually decide whether the drug can be marketed.

Although the pharmaceutical company pays for the clinical trial, it does not conduct the trial itself — partly to avoid an appearance of impropriety and partly for the sake of efficiency. The company hires an accredited *contract research organization* to organize and manage the clinical trial. This trial organizer hires

[6] Our notion of state is influenced by the theory of abstract state machines [11] but we do not presume that the reader is familiar with abstract state machines.

sites, like hospitals or clinical labs, to take part in the clinical trial. A site finds appropriate patients and designates physicians, bioscientists, etc., to work on the clinical trial. We presume that the clinical trial employs cloud computing. Patient records and other data relevant to the trial are kept in the cloud, and access to them is controlled by a special key manager.

The conduct of a clinical trial involves the interaction of a great many entities, known as principals in the context of access control. These include government agencies (e.g. FDA), the pharmaceutical company, the trial organizer, the sites, the physicians, the bioscientists, the auditors, and others [10]. A complete description of the policies of all these principals is a major project which we do not undertake here; rather, we use a small part of such a description as an illustration of how policies are handled in DKAL.

2.2 Policies

The policies we use for our example are the ones involved when a physician wants to read the record of a particular patient. These records are stored in the cloud, indexed by patient ID numbers, and, as indicated above, access to patient records is controlled by a key manager. This key manager's role is to check that an access request is legitimate and, if it is, to reveal the necessary cryptographic key to the requester. The legitimacy of a physician's request for the record of a particular patient depends on several factors. We deal with the simple situation where the physician is requesting access to the record of one of his own patients in the trial. Some information in a patient's record should remain inaccessible even to his trial physician, for example, whether the patient is receiving the drug being tested or a placebo. For brevity, we use "record" to mean the part of a patient's record that should be accessible to his trial physician. (Real policies would also have to cover the case where, in an emergency, a physician — any physician — attending the patient may need the full record, including the information that is ordinarily inaccessible.) The key manager does not simply accept the physician's word that this is his patient but requires certification from the site (in this case a hospital or clinic) that assigned that patient (or, more precisely, that patient's ID) to this particular physician. But the key manager may know only the organizers of trials, not the sites they hire. The key manager therefore requires additional certification, from the trial organizer, that the site in question is a legitimate site, is working on this trial, and is authorized to assign, to a physician of its choice, the patient ID number involved in the physician's request.

It is generally desirable for the key manager to be almost stateless, i.e., to remember very little information. He needs to know the trial organizers and to have a current revocation list (in case an organizer, site, or physician has lost some rights), but he should not have a large database. The less information the key manager keeps, the less the cloud's customers will worry about their information being compromised by bribed technicians, court subpoenas, and similar disasters. Furthermore, the scalability of the system is improved by keeping the key manager's state (including his cache) small.

Thus, our scenario features four of the principals involved in a particular clinical trial called `Trial1`:

- `Org1`, the contract research organization conducting `Trial1`.
- `Site1`, one of the sites hired by `Org1` for `Trial1`.
- `Phys1`, one of the physicians working at `Site1` on `Trial1`.
- `KeyManager` for `Trial1`.

We shall discuss and formalize in DKAL the policies that these principals should follow so that, when an authorized physician wants to get a patient record, he gets a key for it, but unauthorized principals don't get keys. (The key manager, who has all the keys, counts as authorized.)

Informally, the essential parts of these policies are as follows. `Org1` sends, to each site selected for `Trial1`, a (cryptographically signed) certification that it has been hired to work on `Trial1` and has been allocated a certain range of patient ID numbers. Furthermore, `Org1` sends each of these sites a signed statement delegating to the site the authority to grant access to patient records whose ID numbers are in that range. `Site1`, in turn, having received its range `[N1,N2]` of patient ID numbers, sends each of its chosen physicians a signed authorization to access records with ID numbers in a certain sub-interval of `[N1,N2]`. Furthermore, `Site1` forwards to its physicians, including `Phys1`, the statement from `Org1` delegating this access authority to `Site1`. When `Phys1` needs to see the record of a patient whose ID number `N` is in `Phys1`'s assigned interval `[P1,P2]`, he sends a request to `KeyManager`, accompanied by the statements from `Org1` and `Site1` granting that authority, plus verifications of the arithmetical facts that the patient ID is in the range assigned to him by `Site1` (so that `Site1`'s authorization applies) and in the range assigned to `Site1` by `Org1` (so that `Org1`'s delegation of authority applies), plus a proof that these arithmetical facts and the two signed statements establish that he has authority, from `Org1`, to access the record in question.

Finally, `KeyManager`'s policy includes verifying proofs sent to him as well as signatures on documents, trusting `Org1` on such authorizations, and sending the correct key to the requester. (For details, see Section 7.)

3 The World of One DKAL Principal

The part of the world seen by a particular principal can be described as follows. First, there are various facts, which we regard as being recorded in some database(s). In what follows, when we refer to databases, we mean collections of relations, not the additional machinery, such as search and query mechanisms, associated with full-scale databases. For brevity, we also stretch the term "database" to include temporary storage. For example, when a physician receives a key from the key manager, he should not, in view of security considerations, record the key in anything as permanent as what is usually called a database. Rather, he should keep a temporary record of it for as long as it takes him to use the key; then he should destroy it. Since this sort of temporary storage will

play the same role as databases in our discussion, and since we don't want to repeatedly write "databases or temporary storage," we let "databases" refer to both of these.

A principal's data are relations in these databases plus some functions, e.g. standard arithmetical operations, given by algorithms. Some of the relations and functions may be public, with the same contents for all principals. Others may be private for a specific principal. Intermediate situations are also possible, where several but not all principals share some relations and functions.

These relations form what we call the *substrate* of a principal. The public part of a substrate would ordinarily also include things generally available in computing systems, such as numbers and the arithmetical operations on them. In our clinical trial scenario, the publicly available substrate would tell, for example, which contract research organizations are conducting which trials. A contract research organization's substrate would tell which sites are to participate in which trials as well as which patient ID numbers have been allocated to each site. The sites, in turn, would have in their substrates information about the physicians they hired and the patient ID numbers assigned to each physician.

A principal can, in general, obtain and update values of functions in his substrate, but this ability may be limited in several ways. In the first place, some entries may be undefined for the principal — for example, record keys that have not been given to him. Let us suppose that principals use a substrate function `Key` for temporary storage of keys. Then this function would ordinarily be defined at only a small number of arguments in any particular principal's substrate (except in the key manager's substrate).

Furthermore, some parts of the substrate, in particular public parts, may be designated as static, i.e., not susceptible to updating

Another part of a principal's world, the part directly concerned with knowledge and communication, is the *infostrate*. It contains the principal's *knowledge assertions*, which incorporate the knowledge initially available to the principal and the knowledge obtained from communication. In addition, it contains knowledge that the principal has deduced from these together with facts from the substrate. (Actually, there is more to the infostrate, including the principal's policies for sending and accepting communications; see Section 6 for a complete description.)

As we shall see below, the mathematical framework of DKAL treats a principal's substrate and infostrate rather differently. Nevertheless, in real-world situations, there may be some arbitrariness in deciding where a particular aspect of a principal's world should reside. Communications are definitely in the infostrate, and public databases are definitely in the substrate, but between these there are less definite situations. Intuitively, if a principal learns some information from a communication and soon uses it for further communication, this would belong in the infostrate, but if the information is recorded, perhaps in a table, for later use, then the table belongs in the substrate. Ultimately, though, decisions about what goes into the substrate and what goes into the infostrate will be governed

by pragmatic considerations including ease of analysis (which may, of course, benefit from conformity to intuition).

3.1 Substrate Functions and Infon Relations

We adopt several conventions concerning substrates and infostrates.

The substrate of any principal is assumed to be a typed, purely functional, first-order structure, except that the functions may be partial[7]. That is, the substrate is a finite system of nonempty sets together with some (possibly partial) functions between them. Each function has a specified number of arguments, of specified types, and a specified type for its output. One of the built-in types is Boolean, with just two elements, `true` and `false`. Although officially a substrate has only functions, not relations, Boolean-valued functions amount to relations. So in effect we have arbitrary (multi-sorted) first-order structures. Constants are, as usual, regarded as 0-ary functions. Note that, since functions are allowed to be partial, there may be ground terms that are not assigned any value by the usual definition.

A participant's infostrate, in contrast, does not involve function symbols but resembles a relational structure, with the same underlying sets (of various types) as the substrate. Instead of relations, though, it has *infon relations* that assign *infons* (rather than truth values) to tuples of elements of the appropriate types. *Infon formulas* are built from infon relations, constants, and variables according to rules that will be spelled out in Section 5. These infon formulas do not have truth values in the usual sense; rather, their values are *infons*, which we think of as items of information. They are not true or false, as in classical logic, but are known or not known by principals. Infons are the entities that can be communicated from one principal to another. Each principal in a DKAL system has a *policy*, including

- what sorts of communications to accept, and from whom,
- what communications to send, and to whom, and
- what changes to make in its state.

All of these specifications can use information from this principal's substrate and from the principal's *knowledge* and *roster*[8].

The knowledge mentioned here arises from the principal's original knowledge assertions, the communications that the principal has received and accepted, and information from the substrate.

[7] For readers familiar with the abstract state machine literature, we point out the following difference. Suppose that f is a partial function undefined at an argument $\bar{a}$. The abstract state machine convention is that $f(\bar{a}) =$ `undef` where `undef` is a first-class element, so that e.g. $f(\bar{a}) = f(\bar{a})$ evaluates to `true`. In a DKAL substrate there is no value for $f(\bar{a})$, nor for any term that includes it as a subterm. In particular, the value of the equation $f(\bar{a}) = f(\bar{a})$ is undefined.

[8] The roster is actually part of the substrate, but we mention it here for emphasis.

The roster consists of the elements of the substrate that the principal is aware of. The official definitions of "knowledge" and "roster" will be given in Sections 4 and 6.

In practice, it is difficult to separate "knowledge" and "policy" as described above. For example, if a principal decides to trust another on some infon (i.e., "if so-and-so tells me x then x") is this trust assertion a matter of knowledge or of policy? Furthermore, it has become traditional in logic-based authorization to use the word "policy" in a broad sense that includes some of what we have called knowledge, at least the explicit knowledge assertions of a principal. Accordingly, we shall from now on use "policy" in this broader sense. In particular, the formalization of our example in Section 7 will include, in the principals' policies, several trust assertions.

A crucial aspect of infons (or, more precisely, of the infon formulas that denote them) is that, in addition to atomic infon formulas (asserting an infon relation of some named substrate elements) and combinations via "and" and "implies", they can also assert that some principal said (or implied) some other infon. In this way, communications from one principal can enter into the knowledge of another.

3.2 Notational Conventions

In this paper, we choose our notations for function names, infon relations, and variables in accordance with the following conventions. All of these symbols are in the typewriter typeface.

Function symbol: String of letters and digits, beginning with a capital letter and containing at least one lower-case letter.

Infon relation: One or more strings of letters, each string starting with a lower-case letter. (The "or more" is to allow writing some arguments between the parts of the infon relation.)

Variable: Nonempty strings of capital letters, possibly followed by digits.

In addition, we use obvious notation for arithmetical operations and propositional connectives.

Here are some examples. Variables in our examples will include `TRIAL`, `SITE`, `N1`, and `PERSON`.

Function names will include constants (recall that these are nullary functions, i.e., taking no arguments), such as `Trial1`, `Org1`, and `Phys1`. They also include unary functions, such as `Org` (as in `Org(TRIAL)`) and `Key` (as in `Key(RECORD)`), and functions of two or more arguments like `Record` (as in `Record(N, TRIAL)`).

Infon relations include the binary `participates in` and ternary `participates in at as physician`, as in `SITE participates in TRIAL` and `PERSON participates in TRIAL at SITE as physician`.

There is a special infon relation, `asinfon`, which takes one argument of Boolean type and returns an infon whose intended meaning is that the argument is true. That is, it converts classical truth values (things that can be true

or false) into infons (things that can be known by principals). An example, using the availability of elementary arithmetic and propositional logic in the substrate, is `asinfon`(`N1` $\leq$ `N` **and** `N` $\leq$ `N2`).

3.3 Disclaimer

Our primary goal in this paper is to describe the communication and inference mechanisms of DKAL. That is, we shall be interested primarily in infostrate phenomena. To better expose these essential ingredients of DKAL, we shall omit from our examples various things that, though important in reality, would only clutter the example and obscure the main points we wish to make. In particular, we shall not deal here with the updates that principals may perform in their substrates. (Some special updates of substrates, namely additions to principals' rosters, occur automatically as an effect of communication; these are included in the discussion in Section 6.)

Principals' policies may evolve. For example, companies may have to change their policies (keeping a log, of course, to show what their policy used to be) when a government agency changes its rules. They may have to delete or modify some of their knowledge assertions, acquire new knowledge assertions, amend some of their communication rules, etc. We postpone to a future treatment the issue of policy evolution, except that we define, in Subsection 6.3, how accepted communications become new knowledge assertions. For the sake of simplicity, we do not treat in this paper deletions or modifications of knowledge or removal of elements from rosters. Nor do we treat explicit additions, deletions or modifications of communication rules or filters.

Furthermore, although DKAL is a typed system, whose types always include the types of Boolean values, of integers, and of principals, and usually include many other application-specific types, we shall omit explicit indications of types in our examples, trusting in the reader's common sense to correctly understand the types of the expressions we write.

4 Substrate

In logic terms, the substrate of a principal is a multi-type (also called multi-sorted) first-order structure where relations are viewed as Boolean-valued functions. Accordingly the Boolean type, comprising two elements `true` and `false`, is ever present. Other ever present types are the types of principals and integers. The substrate vocabulary contains the names of the substrate functions. Constants are (the names of) nullary functions. Terms over the substrate vocabulary are produced as usual (in first-order logic) from variables by means of (repeated applications of) function names. In particular, constants are terms.

It is convenient to see the substrate as a collection of (types and) partial functions. For instance, the substrate of principal `Org1` contains a binary function `Status(SITE,TRIAL)` one of whose values is `Unnotified`. The intended meaning of `Unnotified` is "chosen to work on this trial but not yet notified". Another

binary function in the substrate of `Org1` is `Patients(SITE,TRIAL)`, which gives the range of patients' identification numbers assigned to the `SITE` at the `TRIAL`.

Some substrate functions, like `Status` or `Patients`, are finite and can be given by tables. In general, a substrate function f is given by an algorithm (which can be just a table lookup). For example, integer multiplication is given by an algorithm and so is integer division. We presume that, given a tuple $\bar{a}$ of arguments of appropriate types, the algorithm first decides whether f is defined at $\bar{a}$. If $f(\bar{a})$ is undefined, the algorithm reports that fact and halts. Otherwise it proceeds to compute $f(\bar{a})$. Thus the algorithm never "hangs". For example, given $(1, 0)$, the algorithm for integer division will report that $1/0$ is undefined.

Equivalently one can think of a substrate as a relational database. Total Boolean-valued functions are relations. Any other function f, of arity j, gives rise to a $(j+1)$-ary relation (the graph of f) with a functional dependency that says that the first j columns determine the last column. For example, the binary function `Status(SITE,TRIAL)` gives rise to a ternary relation, with attributes for sites, trials and statuses, where the values of `SITE` and `TRIAL` determine the value of `STATUS`.

4.1 Canonic Names, and Term Evaluation

We presume that every element in the substrate of a principal A has a canonic name that is a ground term. We identify these elements with their canonic names. It is presumed that canonic names are obviously recognizable as such. For example, A's canonic name for a principal P could be a record identifying P, e.g. via his public cryptographic key, and giving his address. The canonic name for an integer could be its binary notation.

> **Quisani**[9]**:** What if the substrates of two principals A and B share an element? Can the two names differ? For example, is it possible that A uses binary notation for integers while B uses decimal?
>
> **Authors:** No. We presume names have been standardized for common substrate elements.
>
> **Q:** I see a possible problem. Elements of different types might have the same canonic name. For example, if you had identified principals by just their cryptographic keys, those could also be the binary notations for integers.
>
> **A:** Canonic names are ground terms and thus have definite types.

To evaluate a ground term t at a state S means to produce the canonic name for the element $[\![t]\!]_S$ denoted by term t at S. The principal may be unable to evaluate a ground term. For example, in a physician's substrate the function `Key(RECORD)` is defined for few, if any, records. For a particular record, say

[9] Quisani is an inquisitive friend of ours.

`Record`(10,`Trial1`) of patient 10 in `Trial1`, the physician may be unable to evaluate the term `Key`(`Record`(10,`Trial1`)). The values of `Key` that a physician can evaluate, i.e., the ones that are defined in his substrate, are (in our scenario) the ones which he has gotten from the key manager (and has recorded and not yet destroyed).

4.2 Roster

Intuitively the *roster* of a principal A is the finite collection of elements known to principal A. The roster splits into *subrosters*, one subroster for every type. The subroster of type T consists of elements of type T. Formally, the subrosters are unary relations.

Unsurprisingly, the Boolean subroster consists of `true` and `false`, and the principal subroster of A contains at least (the canonic name of) A.

Roster may not be closed under substrate functions. For example, even if the substrate has a principal-to-principal function Manager, the principal subroster may contain some employee `John` but not contain (the canonic name for) `Manager(John)`. In particular the integer subroster contains only finitely many integers.

5 Logic

5.1 Infons

Traditionally, in mathematical logic, declarative statements represent truth values. In DKAL, as explained in Section 3, we view declarative statements as containers of information and call them infons [12]. This has a relativistic aspect. Indeed, the world of DKAL consists of communicating principals, and each principal lives in his own state. As a result, one question, whether a given statement φ is absolutely true or absolutely false, is replaced with as many questions as there are principals: Does a given principal know φ?

In the first part of the present section, we fill in details about infons, making precise what was outlined in Section 3.

Recall the connection between truth values and infons, given by the `asinfon` construct.

Q: I guess a principal A knows `asinfon`(b) if and only if the Boolean value b is true.

A: That is the idea but one has to be a bit careful. It may be infeasible for the principal A to carry out the evaluation. For example,

$$3^{3^{3^3}} + 8^{8^{8^8}} \texttt{ is prime}$$

has a well defined Boolean value but A may not know it.

5.2 Infons as formulas

Traditionally, statements are represented by formulas. In the previous DKAL papers, we treated infons as objects and represented infons by terms; in particular we had infon-valued functions. In the meantime we realized that infons can be naturally represented by formulas, and that is what we do in this paper.

> **Q:** Since you are interested in statements like "a principal A knows infon formula φ", you may want to work in epistemic logic where formula $K_A(\varphi)$ means exactly that A knows φ.
>
> **A:** It is convenient to have the knowledge operator K_A implicit when you work in the space of A.
>
> **Q:** Explicit knowledge operators allow you to nest them, e.g. $K_A K_B \varphi$.
>
> **A:** Allow or provoke? In the world of DKAL, A never knows what B knows, though A may know what B said.

A principal's vocabulary is the union of his substrate and infostrate vocabularies.

Substrate Vocabulary Recall that a principal's substrate is a many-typed structure of sets (interpretations of type symbols) and partial functions (interpretations of function symbols). We assume that the number of these symbols is finite. Every function symbol f has a type of the form $T_1 \times \ldots \times T_j \to T_0$ where j is the arity of f and the T_i are type symbols. Constants are nullary function symbols. The types of Boolean values, integers and principals are always present. Recall that relations, e.g. the order relation $\leq$ on integers, are treated as Boolean-valued functions.

For every type, there is an infinite supply of variables of that type. We use numerals and standard arithmetical notation. In addition we use strings for the names of variables and functions; see Section 3 for our conventions concerning these strings and for examples of their use.

Infostrate vocabulary Traditionally, relations take values `true` and `false`. *Infon relations*, which constitute, a principal's infostrate vocabulary, are similar except that their values are infons. Their arguments come from the substrate of the principal. Each infon relation has a particular type for each of its argument places. So infon relations assign infons to tuples of elements of the appropriate types.

Atomic infon formulas Prefix or infix notation is used for infon relations. Here are some examples.

```
SITE participates in TRIAL
SITE is allocated patients N1 to N2 in TRIAL
asinfon(N1 ≤ N and N ≤ N2)
```

Composite infon formulas These are built from atomic formulas by means of conjunction $\wedge$, implication $\rightarrow$, and two unary connectives p `said` and p `implied` for every term p of type principal. To make complicated implications easier to read, $\alpha \rightarrow \beta$ can be written as

if
α
then
β

Here's an example:

if
`asinfon(N1` $\leq$ `N` **and** `N` $\leq$ `N2)` **and**
`SITE implied PERSON may read Record(N,TRIAL)`
then
`Org1 implied PERSON may read Record(N,TRIAL)`

Syntactic sugar Implications

$(p$ `said` $\varphi) \rightarrow \varphi$,
$(p$ `implied` $\varphi) \rightarrow \varphi$

are abbreviated to

p `is trusted on saying` φ,
p `is trusted on implying` φ.

In fact, these abbreviations are so common that they are often further abbreviated to p `tdonS` φ and p `tdonI` φ respectively. We note that a similar abbreviation, `A controls s` for `(A says s)` $\rightarrow$ `s`, was used in [1, page 7].

5.3 Logics and Theories

In [13], two of us introduced and investigated propositional infon logics, in particular *primal* infon logic where the derivation problem, whether a given formula φ follows from a given set Γ of formulas, is solvable in linear time. Primal infon logic with (universally quantified) variables is obtained from propositional primal logic by adding the substitution rule. The derivation problem for primal logic with variables remains feasible (though not linear time) [6,7]. Furthermore, security policies involve arithmetic, etc. Instead of pure infon logic, we should be working with infon theories.

Separation of concerns The previous versions of DKAL had a built-in version of infon logic. Here we work with infon theories, and — to separate concerns — we view infon theory as a parameter. In the rest of the article:

– infon logic is primal logic with variables or some extension of it that is a sublogic of classical first-order logic, and

– infon theory includes the theory of arithmetic in the infon logic.

Infon theory is to be given by some axioms. There is considerable flexibility in the choice of axioms. In particular there is no need to minimize the system of axioms. In fact, for our purposes a rich system of axioms may be more appropriate. It is, however, necessary that the axioms be efficiently recognizable.

When we say that a set Γ of infon formulas entails an infon formula φ we mean that Γ entails φ in the fixed infon theory.

From time to time, a DKAL principal may need to compose a proof in the infon theory. In difficult cases he may use an automated or interactive prover to compose proofs. In any case, the methods used for proof composition, even if formally specified, belong to a level of abstraction lower than what we consider in this paper.

Remark 1. The current implementation of DKAL [9] employs primal infon logic with variables and uses the SQL database engine. The infon theory uses ground arithmetical facts that are checked by means of the SQL engine.

Remark 2. In this and the previous subsections, we speak about infon theory used by principals. A more powerful theory may be needed for the analysis of security policies and their interaction.

5.4 Justifications

1. If φ is an infon formula in one of the following two forms
   1a. `A said` α`,`
   1b. $\beta \rightarrow$ `A implied` α,
   then a cryptographic signature of principal `A` under (a strong hash of) the whole formula φ, accompanied with the canonic name of `A`, is a *justification* for φ. The signature is specific for (the hash of) φ; thus changing the formula would (with overwhelming probability) make the signature invalid[10].
2. Let φ be an arbitrary infon formula. A *justification* for φ is a derivation of φ in the fixed infon theory from hypotheses of the form 1a or 1b together with justifications by signatures for all the hypotheses[11].

Q: You mention a cryptographic signature under a formula φ. Do you mean that φ needs to be encrypted?

A: Not necessarily. This depends on the purpose and may vary even within one application.

Q: In connection with case 1b, what if `A` wants β to be checked by a particular principal `B`, essentially delegating the matter to `B`?

[10] In case 1a, our notion of justification is similar to Binder's notion of certificate [8].

[11] Proof-carrying communications, in higher-order logic, were used by Appel and Felten [3]. In contrast, the current implementation [9] of DKAL uses logic that is decidable and in fact feasible.

A: He may replace β with `(B said` β`)` or with `(B implied` β`)` depending on whether he intends to allow `B` to delegate the matter further.

Q: What about `asinfon` formulas? How can they be justified?

A: Only by a derivation in the infon theory, as in item 2 above. Some of them may happen to be axioms.

Q: So many `asinfon` formulas may not be justifiable.

A: That's right. First, the substrates of different principles may disagree. Second, even in the case when only public relations are involved, where all principals agree, things might change over time. But the infon theory should be global (for all principals) and permanent (for all time).

6 Infostrate

The infostrate of a principal A is determined by the substrate and these:

- Knowledge assertions,
- Communication rules,
- Filters.

6.1 Knowledge

At each state, each principal has a finite set of knowledge assertions, which are infon formulas in the vocabulary of the principal enriched by his roster. This set is composed of original knowledge assertions that the principal always had and other knowledge assertions that he acquired from communications. In our example, `Site1` has an original knowledge assertion

```
Org(TRIAL) is trusted on saying SITE participates in TRIAL
```

and he learns

```
Org1 said Site1 participates in Trial1
```

A knowledge assertion of a principal may have free variables. Its instances in a particular state are obtained by replacing variables with the principal's roster elements of the corresponding types.

Knowledge assertions constitute one source of the principal's knowledge. Another source is his substrate. If the principal successfully evaluates to `true` a ground substrate term t of Boolean type then he learns `asinfon`(t). Further, the principal may use the fixed infon theory to derive additional infon formulas from his knowledge assertions and substrate facts. We shall say that a principal *knows* an infon formula φ at a state S if there is a feasible derivation of φ in the infon theory from that principal's knowledge assertions and substrate information.

Q: This "feasible" is vague. If it takes the principal two minutes to deduce φ, fine. But if φ happens to be P=NP and if it happens to be deducible with a few years' effort, must he deduce that?

A: Well, there may be situations where even two minutes is too long a time. But your point is well taken. The decision how much work and time to put into deducing φ depends on the scenario at hand.

6.2 Communication

The infostrate of a principal A may (and typically does) contain communication rules. These rules may cause A to send communications to various principals. We presume in this paper that communications automatically come with identifications of the senders. The recipients filter their communications; a communication may be accepted or rejected.

Communications of one principal to another may or may not be accompanied by a justification of their contents. In the future, a communication accompanied by a justification of its content will be called *justified*. As we already said in the introduction, both kinds of communications are important but, in this paper, we restrict attention to justified communication.

The work in this paper is independent of the mechanisms by which principals communicate. The mechanisms could include message passing, shared memory, bulletin boards, etc. In particular, even when one principal sends a communication to just one other principal, we do not presuppose that it is necessarily message passing.

The communication rules of any principal A have two possible forms:

(Send) **if** π
then send [justified] to r
φ

and

(Post) **if** π
then post [justified] to x
φ

Here π and φ are infon formulas, r is a principal term, and x is a variable of type principal not occurring in π. We call π the *premise*; both r and x are *recipient* terms; and φ is the *content* of the rule. Further, **justified** is optional.

Q: What's the difference between sending and posting, apart from the restriction, in posting, that x is a variable not in the premise?

A: Intuitively, posting does not require the sender to know all the recipients. Formally, the difference will become clear when we talk about how to instantiate variables.

The variables that occur in a communication rule split into two categories. The variables that occur in the premise or in the recipient term are intended to be instantiated by principal A, the sender, before he sends anything out. In the case of (Send), all these variables are instantiated by elements of A's roster. In the case of (Post), the variables in the premise π are instantiated by elements of A's roster but the recipient variable x is thought of as instantiated by all elements of its type in A's substrate.

Q: Now the difference is clear, but what is this "thought of"? That seems like an evasion. I think I see the problems that you try to evade. One is that there could be a huge number of elements of the type of x. Another is that, even if there are only a few elements, A may not know them all.

A: You are right. These are the problems. We don't intend for A to actually send individual messages to all these recipients, whether there are many of them or few. Rather, he should post the content φ where the intended recipients can read it or download it, etc.

For simplicity, in the rest of the paper, we consider only (Send) rules, always justified. Our example in Section 7 will have only such rules.

The variables of a communication rule that occur only in the content are called *verbatim* variables. They are sent out uninstantiated, verbatim, and will be instantiated by the receivers using elements of their rosters.

Q: Obviously a variable v of type T is instantiated by elements of type T. Suppose that v is one of those verbatim variables. Clearly the sender has the type T; otherwise he would never have such a variable. What about the receiver? He also needs T in order to instantiate v properly. How does the sender know that the receiver has T?

A: The language does not guarantee that the sender knows this. If the receiver does not have T, the communication is unintelligible to him and therefore rejected[12].

Q: OK, let us assume that the sender and receiver both have type T in their vocabularies. Should it be the same type in the semantic sense; that is, should T have the same elements in both substrates?

A: Not necessarily. Consider the type Secretary. The secretaries on the sender side and those on the receiver side may be different but the sender may consciously have in mind the receiver's secretaries.

Q: What if the receiver has type T but it is not at all the type that the sender had in mind? For example, being the government, the receiver thinks that "secretary" means a member of the president's cabinet.

[12] The mechanism of rejection is filters; see below. A filter admits only those communications that match some pattern. An unintelligible communication will not match any pattern.

A: Too bad. This communication will be misunderstood.

Q: OK, this clarifies the situation for types. Something else bothers me. Suppose the content contains $f(v)$ where v is a verbatim variable or, more generally, $f(t)$ where t contains a verbatim variable. The sender can't evaluate this, so the recipient must. How does he interpret the symbol f?

A: The situation for such function symbols is the same as for types of verbatim variables. The recipient must have f in his vocabulary, with the same type as the sender's f; otherwise the communication is unintelligible. And if the recipient's interpretation of f is seriously different from what the sender intended, then the communication may be misunderstood.

In a state S of principal A, the communication rule (Send) may result in a number of communications to various principals. A *sender* substitution ζ for (Send) instantiates the variables in the premise π and the recipient term r (that is, all non-verbatim variables) with canonic names from the current roster. If A can evaluate the infon formula $\zeta(\pi)$ and knows the resulting infon, and if he can evaluate $\zeta(r)$, then he applies ζ to the content φ and evaluates the ground terms in the resulting infon formula. If the evaluation fails because of an undefined term, that instance of (Send) produces no communication. Suppose the evaluation succeeds, and let's call the resulting infon formula $\zeta(\varphi)$. Thus $\zeta(\varphi)$ is obtained from φ by substitution followed by evaluation. If A is able to construct a justification for $\zeta(\varphi)$ then communication $\zeta(\varphi)$, complete with a justification, is sent to principal $\zeta(r)$.

Q: This "able to construct" sounds vague to me.

A: You are right. It is vague in the same sense as "feasible". The effort devoted to constructing justifications would depend on the scenario at hand.

Q: OK. But now what if the sender has two (or more) justifications for the content φ? Which one does he send?

A: DKAL imposes no restrictions on the choice of justification.

For some applications, we would want to allow, in the content φ of (Send), *verbatim function symbols*. These would not be evaluated during the computation of $\zeta(\varphi)$, even if all their arguments have been successfully evaluated. Rather they would be sent as is, to be evaluated later. For example, an executive A tells another executive B: "Have your secretary call my secretary to set up an appointment". The first occurrence of "secretary" needs to be verbatim if A does not know B's secretary. The second occurrence of "secretary" should not be verbatim. Verbatim function symbols are introduced in [14]. We will not use them in our example in Section 7. Note, however, that our communication rules do contain occurrences of function symbols that the sender will not evaluate, namely ones whose arguments cannot be evaluated because they involve verbatim variables.

Speeches A send rule of principal A of the form

if π
then send [justified] to r
A `said` ψ

may be abbreviated to

if π
then say [justified] to r
ψ

6.3 Filters

A principal filters his incoming communications. He may have several filters, and a communication will be accepted provided at least one of the filters admits it. In the world of Evidential DKAL where all communications are justified, a filter has the form

(Filt) **if** ρ
then accept if justified from s
Ψ

Here ρ is an infon formula, called the *premise*, and s is a principal term, called the *sender* term. Further, Ψ is a pattern, which is defined as follows. An *atomic pattern* is an atomic infon formula or an infon variable. *Patterns* are either atomic patterns or built from them exactly as composite infon formulas are built from atomic infon formulas, that is, by means of conjunction, implication, p `said`, and p `implied`.

The part "if justified" in (Filt) means that the communication must be accompanied by a justification which the recipient should verify before accepting the communication.

Suppose principal `B` receives a communication φ from another principal `A`. First, `B` adds (the canonic name of) `A` to his roster, unless it is already there. (`B` may want to remove `A` from the roster later on, but he should have `A` on the roster at least while processing φ.) Second, for each of his filters F, the principal `B` tries instantiating the variables in the premise ρ and the sender term s, using elements of his roster. He wants an instantiation η such that he knows $\eta(\rho)$ and such that $\eta(s)$ evaluates to `A`. For each instantiation η that succeeds in this sense, he checks whether the communication φ matches the pattern $\eta(\Psi)$ in the following sense. He can get φ from $\eta(\Psi)$ by replacing infon variables with infon formulas and replacing the remaining variables with terms of the same types. Third, for each instantiation η that has succeeded so far, `B` verifies the justification of φ. If the verification also succeeds, then he adds φ to his knowledge assertions, and he adds to his roster all the canonic names that he can obtain by evaluating ground terms that occur in φ.

Q: Is verifying a justification also a vague notion?

A: No, the recipient must really verify the justification. Notice though that the work required is bounded in terms of the size of the communication φ plus the justification.

Q: Right. In fact, it seems to be bounded just in terms of the justification.

A: Not quite. Recall that the justification could be a signature on a hash of φ. Verifying it would require computing the hash of φ from φ.

Q: You showed how filters look in Evidential DKAL. I guess that in general not-necessarily-evidential DKAL they look the same except that **if justified** may be omitted and then the recipient does not have to check a justification.

A: Not quite. There is an additional difference. The content φ does not automatically become a knowledge assertion of the recipient. The filter may give other ways to accept a communication. For example, the 911 operators should not automatically believe every anonymous report they get, but they should not completely ignore a report either.

In the present paper, for brevity, we make the following simplifying assumption. Each principal has exactly one filter, namely

if `asinfon(true)`
then accept if justified from x
  Ψ

where x is a principal variable and Ψ is an infon variable. In other words, all communications in the vocabulary of the recipient are accepted subject to the verification of justification.

7 Clinical Trial Scenario: Policies

The clinical trial scenario was described informally in Section 2. Here we formulate the relevant parts of the policies of `Org1`, `Site1`, `Phys1` and `KeyManager` in DKAL. For brevity, we do not declare the types of the variables, leaving it to the reader's common sense to interpret variables correctly. We also remind the reader that we use "record" to mean the part of a patient's medical record that should be accessible to his trial physician; `Record(N,TRIAL)` refers to the record, in this sense, of the patient with ID number `N`.

7.1 Policy of Org1

```
if
   asinfon(Org(TRIAL) = Org1)
   asinfon(SiteStatus(SITE,TRIAL) = Unnotified)
   asinfon(SitePatients(SITE,TRIAL) = [N1,N2])
then
   say justified to SITE
      SITE participates in TRIAL
      SITE is allocated patients N1 to N2 in TRIAL
   send justified to SITE
      if
         asinfon(N1 ≤ N and N ≤ N2)
         SITE implied PERSON may read Record(N,TRIAL)
      then
         Org1 implied PERSON may read Record(N,TRIAL)
```

Q: You surely intend that the sites, chosen by `Org1` to participate in a particular trial, are notified about that just once. But what prevents `Org1` from executing the rule over and over again?

A: Currently the rule has one **say** and one **send** commands. To make our intention explicit, we can add a third command, namely an assignment that changes the value of `SiteStatus(SITE,TRIAL)`.

Q: Changes it to `Notified`? Or maybe the new value should be `Pending`, which later is changed again to reflect the site's reaction.

A: That's the thing. Communication rules may have side effects, in the form of substrate updates, which detailed policies would specify. That would involve decisions that are irrelevant to us here. Accordingly, we omit in this example the explicit updates of the substrate[13]. Concerning `Org1`, we just assume that, for each clinical trial and each site chosen by `Org1` to participate in the trial, the communication rule is executed exactly once.

Q: Concerning the syntax of the rule, I see that the two keyword pairs **if . . . then** are used for two different purposes.

A: Yes, one **if** starts the premise of the rule, and the corresponding **then** starts the body. Another pair is syntactic sugar in an infon formula; it is used to render an implication in a form that is easier to read. As long as we're talking syntax, let's also clarify another point. When two or more infon formulas are exhibited, one after the other without connectives between them, we mean the conjunction. The premise above is the conjunction of three infon formulas. The content of the **say** is the

[13] Related issues are addressed in [4].

conjunction of two infon formulas, and the premise of the implication is the conjunction of another pair infon formulas.

Q: Now let me look into the necessary justifications. Isn't it true that all that is needed is two signatures of `Org1`?

A: Yes. One signature goes with the **say justified** command, and the other with the **send justified** command. In the second case, a signature suffices because the content has the form $\alpha \rightarrow$ `Org1 implied` β.

7.2 Policy of Site1

```
Org(TRIAL) is trusted on saying
   SITE participates in TRIAL
Org(TRIAL) is trusted on saying
   SITE is allocated patients N1 to N2 in TRIAL

if
   Site1 participates in TRIAL
   Site1 is allocated patients N1 to N2 in TRIAL
   asinfon(PhysStatus(PHYS,Site1,TRIAL) = Unnotified)
   asinfon(PhysPatients(PHYS,Site1,TRIAL) = [P1,P2])
   asinfon(N1 ≤ P1 and P2 ≤ N2)
then
   say justified to PHYS
      PHYS participates in TRIAL at Site1 as physician
      PHYS is allocated patients P1 to P2 in TRIAL at Site1
   send justified to PHYS
      if asinfon(P1 ≤ N and N ≤ P2)
      then Site1 implied PHYS may read Record(N,TRIAL)
   send justified to PHYS
      if
         asinfon(N1 ≤ N and N ≤ N2)
         Site1 implied PERSON may read Record(N,TRIAL)
      then
         Org(TRIAL) implied PERSON may read Record(N,TRIAL)
```

Q: It is presumed I guess that, for each trial that `Site1` participates in, `Site1` chooses appropriate physicians and then notifies them just once.

A: Exactly.

Q: Now again let me look into the three justifications. The first two are simply `Site1`'s signatures. What about the third?

A: Here `Site1` is simply forwarding, literally, what it received from `Org1`, with the original signature of `Org1`. That's it.

Q: Literally? One discrepancy is that there was a variable SITE in Org1's communication rule. I do not see it in the forwarded version.

A: That variable occurred also in the premise of Org1's rule. So it was instantiated. The actual communication from Org1 to Site1 has Site1 in its place.

Q: There is another discrepancy. The last line of Site1's rule has Org(TRIAL) where Org1's policy has Org1.

A: The variable TRIAL occurs also in the premise of Site1's rule, so it will be instantiated. Let's suppose that Org1 nominated Site1 to participate in Trial1 and assigned to it a range N1 to N2 of patient IDs. We presume that function Org is public. In particular, Site1 can evaluate Org(Trial1) to Org1. Accordingly, it trusts Org1 on saying that Site1 participates in Trial1 and is allocated the range N1 to N2. So, in its communication rule, Site1 can instantiate TRIAL to Trial1. Then Org(TRIAL) that you asked about will be instantiated to Org(Trial1) and will be evaluated to Org1.

7.3 Policy of Phys1

```
SITE is trusted on saying
   PHYS participates in TRIAL at SITE as physician
SITE is trusted on saying
   PHYS is allocated patients P1 to P2 in TRIAL at SITE

KeyManager is trusted on saying
   key of Record(N,TRIAL) is K


if
   Phys1 participates in TRIAL at SITE as a physician
   Phys1 is allocated patients P1 to P2 in TRIAL at SITE
   asinfon(P1 ≤ N and N ≤ P2)
   asinfon(NeedInfo(N))
then
   send justified to KeyManager
      Phys1 said Phys1 requests to read Record(N,TRIAL)
      Org(TRIAL) implied Phys1 may read Record(N,TRIAL)
```

Q: I understand that Phys1 sends to KeyManager the conjunction of two infon formulas. For the first conjunct, you could have used the **say** command with the content Phys1 requests to read Record(N,TRIAL). Wouldn't that look more natural?

A: Maybe, but it would require separating the two conjuncts. If the implementation then sends them separately, the `KeyManager` would have to remember one conjunct until he gets the other.

Q: All right. Now let's look at the justification. The variables `N` and `TRIAL` are instantiated, say to 10 and `Trial1`, and of course `Org(TRIAL)` then evaluates to `Org1`. `Phys1` needs to justify a conjunction which he can do by justifying both conjuncts. The first conjunct is easy. It can be justified by `Phys1`'s signature. How does he justify the second conjunct?

A: He needs to deduce the second conjunct in the fixed infon theory from the following infon formulas. First there is `Org1`'s signed statement

```
(1)   if      asinfon(N1 ≤ N and N ≤ N2)
              SITE implied PERSON may read Record(N,TRIAL)
      then    Org1 implied PERSON may read Record(N,TRIAL)
```

Second there is `Site1`'s signed statement

```
(2)   if      asinfon(P1 ≤ N and N ≤ P2)
      then    Site1 implied PHYS may read Record(N,TRIAL)
```

Finally, there are the arithmetical facts (with no variables!)

```
(3)   asinfon(N1 ≤ 10 and 10 ≤ N2)
      asinfon(P1 ≤ 10 and 10 ≤ P2)
```

for which `Phys1` can easily supply proofs in the infon theory.

Q: Aren't `N1, N2, P1, P2` variables?

A: They have been instantiated but, by slight abuse of notation, we continue to use the same symbols.

Q: In order to combine (1)–(3), I guess, `Phys1` will need to apply the substitution rule of infon logic: `N` becomes 10, `SITE` becomes `Site1`, `PERSON` and `PHYS` become `Phys1`, and `TRIAL` becomes `Trial1`.

A: Right. And once that's done, all that remains is to apply modus ponens a few times.

7.4 Policy of KeyManager

```
if
   PERSON said PERSON requests to read Record(N,TRIAL)
   Org(TRIAL) implied PERSON may read Record(N,TRIAL)
   asinfon(Key(Record(N,TRIAL)) = K)
then
   say justified to PERSON
      key of Record(N,TRIAL) is K
```

Q: Why require that `KeyManager`'s communication be justified? The person, say `Phys1`, gets the key. It is more important that the key is encrypted.

A: Yes, the key should be encrypted. One can argue that a signature of `KeyManager` under the infon is useful. In any case, we agreed to restrict attention to justified communication in this paper.

Q: It seems artificial that the key manager sends an infon to `Phys1` rather than just the key.

A: It makes sense to indicate what the key is for, especially because `Phys1` might have asked for several keys at once, and the key manager obliged.

8 Future work

It is about time that logic-based authorization graduates from academic labs to industrial applications. There is a great need for specification-level authorization, and many organizations, e.g. the technical committees for TSCP [19] and for XACML [20], understand that. Other potential users of these methods may not yet even be aware of them. Logic-based authorization should be prepared to meet the evolving needs of the customers. The languages should be both sufficiently rich and user-friendly.

We are actively working on a number of extensions of DKAL. Here we mention just three of them.

One pressing issue for DKAL is the need to manage substrates in a high-level transparent way. Essentially the same issue is discussed in [4]. We intend to use the abstract state machine (ASM) approach that has been used for some time at Microsoft [16,18] and elsewhere. We view a principal's substrate as an ASM state. The notion of communication rule should be expanded to a notion of rule that allows, in addition to **send** and **post** commands, arbitrary ASM rules to update the substrate.

Similarly, we need to expand the communication mechanism of DKAL so that it can be used to modify not only a principal's knowledge assertions but also his communication rules and filters (and ASM rules).

And, as we mentioned in the introduction, we also need to bring non-justified communication into the current framework.

References

1. Abadi, M., Burrows, M., Lampson, B., Plotkin, G.: A Calculus for Access Control in Distributed Systems. ACM Trans. on Programming Languages and Systems, 15:4, 706–734 (1993)
2. Aikhenvald, A.Y.: Evidentiality. Oxford University Press (2004)
3. Appel, A.W., Felten, E.W.: Proof-Carrying Authentication. In: 6th ACM Conference on Computer and Communications Security, 52–62 (1999)
4. Becker, M.Y.: Specification and Analysis of Dynamic Authorisation Policies. In: 22nd IEEE Computer Security Foundations Symposium, 203–217 (2009)
5. Becker, M.Y., Fournet, C., Gordon, A.D.: SecPAL: Design and Semantics of a Decentralized Authorization Language. Journal of Computer Security 18:4, 597–643 (2010)
6. Beklemishev, L., Gurevich, Y.: Infon Logic (tentative title). In preparation
7. Blass, A., Gurevich, Y.: Hilbertian Deductive Systems and Horn Formulas (tentative title). In preparation
8. DeTreville, J.: Binder, a Logic-Based Security Language. In: IEEE Symposium on Security and Privacy, 105–113, (2002)
9. DKAL at CodePlex. `http://dkal.codeplex.com/`, viewed July 6, 2010.
10. Shayne Cox Gad (ed.): Clinical Trials Handbook. Wiley (2009)
11. Gurevich, Y.: Evolving Algebra 1993: Lipari Guide. In: Specification and Validation Methods, Oxford University Press, 9–36 (1995)
12. Gurevich, Y., Neeman, I.: DKAL: Distributed-Knowledge Authorization Language. In: 21st IEEE Computer Security Foundations Symposium, 149–162 (2008)
13. Gurevich, Y., Neeman, I.: Logic of Infons: The Propositional Case. ACM Trans. on Computational Logic, to appear. See `http://tocl.acm.org/accepted.html`.
14. Gurevich, Y., Neeman, I.: DKAL 2 — A Simplified and Improved Authorization Language. Microsoft Research Technical Report MSR-TR-2009-11 (2009)
15. Gurevich, Y., Roy, A.: Operational Semantics for DKAL: Application and Analysis. In: 6th International Conference on Trust, Privacy and Security in Digital Business, Springer LNCS 5695, 149–158 (2009)
16. Gurevich, Y., Rossman, B., Schulte, W.: Semantic Essence of AsmL. Theoretical Computer Science 343:3, 370–412 (2005)
17. Lampson, B., Abadi, M., Burrows, M., Wobber, E.P.: Authentication in Distributed Systems: Theory and Practice. ACM Trans. on Computer Systems 10:4, 265–310 (1992)
18. Spec Explorer: Development `http://msdn.microsoft.com/en-us/devlabs/ee692301.aspx` and research `http://research.microsoft.com/en-us/projects/specexplorer/`, viewed July 6, 2010
19. TSCP: Transglobal Secure Collaboration Program, `http://tscp.org/`, viewed July 6, 2010
20. XACML: Extensible Access Control Markup Language, `http://xml.coverpages.org/xacml.html`, viewed July 6, 2010

Engineering and Software Engineering

Michael Jackson

Department of Computing
The Open University
Milton Keynes MK7 6AA
United Kingdom

Abstract. The phrase 'software engineering' has many meanings. One central meaning is the reliable development of dependable computer-based systems, especially those for critical applications. This is not a solved problem. Failures in software development have played a large part in many fatalities and in huge economic losses. While some of these failures may be attributable to programming errors in the narrowest sense—a program's failure to satisfy a given formal specification—there is good reason to think that most of them have other roots. These roots are located in the problem of software engineering rather than in the problem of program correctness. The famous 1968 conference was motivated by the belief that software development should be based on "the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering." Yet after forty years of currency the phrase 'software engineering' still denotes no more than a vague and largely unfulfilled aspiration. Two major causes of this disappointment are immediately clear. First, too many areas of software development are inadequately specialised, and consequently have not developed the repertoires of normal designs that are the indispensable basis of reliable engineering success. Second, the relationship between structural design and formal analytical techniques for software has rarely been one of fruitful synergy: too often it has defined a boundary between competing dogmas, at which mutual distrust and incomprehension deprive both sides of advantages that should be within their grasp. This paper discusses these causes and their effects. Whether the common practice of software development will eventually satisfy the broad aspiration of 1968 is hard to predict; but an understanding of past failure is surely a prerequisite of future success.

Keywords: artifact, component, computer-based system, contrivance, feature, formal analysis, normal, operational principle, radical, specialisation, structure.

1 Software Engineering Is about Dependability

The aspiration to 'software engineering' expresses a widely held belief that software development practices and theoretical foundations should be modelled on those of the established engineering branches. Certainly the record of those branches is far from perfect: the loss of the space shuttle Challenger, the collapse of the Tacoma Narrows bridge, and the Comet 1 crashes due to metal fatigue are merely the most notorious of

S. Nanz (ed.), *The Future of Software Engineering*,
DOI 10.1007/978-3-642-15187-3_6,

many engineering failures. But—rightly or wrongly—these failures are seen as local blemishes on a long record of consistently improving success. Failures of software projects and products seem to be the rule rather than the exception. Sardonically, we compare the frustrations of using Windows with the confident satisfactions of using a reliable modern car. At a more serious level, software engineering defects often play a large part in critical system failures. A nuclear power station control system shut down the reactor when a software update was installed on a connected data acquisition system [12]. In the years 1985–1987, software defects of the Therac-25 radiotherapy machine caused massive radiation overdoses of several patients with horrendous results [13]. Nearly twenty five years later, more modern radiation therapy machines were involved in a much larger number of very similar incidents [3]. Software defects, of the kind that were responsible for the Therac disasters, were a major contributory factor in many of these contemporary incidents.

These and the multitudinous similar failures reported in the Risks Forum [21] are evidence, above all, of the lack of dependability [10] in the products of software engineering. Regrettably, we are not astounded—perhaps we are no longer even surprised—by these failures. As software engineers we should place dependability foremost among our ambitions.

2 A Software Engineer's Product Is Not the Software

A common usage speaks of the physical and human world as the 'environment' of a computer-based system. This usage is seriously misleading. The word 'environment' suggests that ideally the surrounding physical and human world affects the proper functioning of the software either benignly or not at all. If the environment provides the right temperature and humidity, and no earthquakes occur, the software can get on with its business independently, without interference from the world.

This is far from the truth. Dijkstra observed [6] that the true subject matter of a programmer is the computation evoked by the program and performed by the computer. In a computer-based system, in which the computer interacts with the physical and human world, we must go further. For such a system, the true subject matter of the software engineer is the behaviour evoked by the software in the world outside the computer. The system's purpose is firmly located in its *problem world*: that is, in those parts of the physical and human world with which it interacts directly or indirectly. Software engineers must be intimately concerned with the problem world of the system whose software they are developing, because it is that problem world, enacting the behaviour evoked and controlled by the software, that is their true product. The success or failure of a radiotherapy system must be judged not by examining the software, but by observing and evaluating its effects outside the computer. Do the patients receive their prescribed doses of radiation, directed exactly at the prescribed locations? Are individual patients dependably identified or are there occasional confusions? Is the equipment efficiently used? So developers of a radiotherapy system must be concerned with the detailed properties and behaviour of every part of the therapy machine equipment, with the positioning and restraint of patients undergoing treatment, with the radiologist's procedures, the oncologist's

prescriptions, the patients' behaviours, needs and vulnerabilities, and with everything else that contributes substantially to the whole system.

3 The Software and Its Problem World Are Inseparable

Twenty years ago Dijkstra [7] argued that the intimate relationship between the software and its problem world could—and should—be dissolved by interposing a formal program specification between them:

> "The choice of functional specifications—and of the notation to write them down in—may be far from obvious, but their role is clear: it is to act as a logical 'firewall' between two different concerns. The one is the 'pleasantness problem,' i.e. the question of whether an engine meeting the specification is the engine we would like to have; the other one is the 'correctness problem,' ie the question of how to design an engine meeting the specification. [...] the two problems are most effectively tackled by [...] psychology and experimentation for the pleasantness problem and symbol manipulation for the correctness problem."

The argument, attractive at first sight, does not hold up under examination. Even for such problems as GCD, the sieve of Eratosthenes, and the convex hull of a set of points in three-dimensional space, the desired firewall is more apparent than real. The knowledge and skill demanded of the software developer are not restricted to the formal symbol manipulations foreseen in the programming language semantics: they must also include familiarity with the relevant mathematical problem world and with a sufficient body of theorems. A candidate developer of a program to "print the first thousand prime numbers" would be conclusively disqualified by ignorance of the notion of a prime number, of integer multiplication and division, and of the integers themselves. Far from acting as a firewall between the program and its problem world, the program specification is inescapably interwoven with the relevant part of the problem world, and tacitly relies on the programmer's presumed prior knowledge of that world.

The firewall notion fails even more obviously for computer-based systems whose chief purposes lie in their interactions with the physical world. The problem world for each such system is a specific heterogeneous assemblage of non-formal problem domains. In any feasible design for the software of such a system the behaviours of the software components and problem world domains are closely intertwined. The locus of execution control in the system moves to and fro among them all, at every level and every granularity. The software components interact with each other at software interfaces within the computer; but they also interact through the problem world. A software component must take account not only of the values of program variables, but also of the states of problem domains outside the computer. The problem domains, therefore, effectively act as shared variables, introducing additional interaction paths between software components. An intelligible formal specification of the system, cleanly separating the behaviour of the software from the behaviour of the problem domains, is simply impossible.

4 A Computer-Based System Is a Contrivance in the World

The products of the traditional branches of engineering are examples of what the physical chemist and philosopher Michael Polanyi calls [19] 'contrivances'. A device or machine such as a motor car, a pendulum clock, or a suspension bridge is a particular kind of contrivance—a physical artifact designed and built to achieve a specific human purpose in a restricted context in the world. A computer-based system is a contrivance in exactly the same sense.

A contrivance has a configuration of characteristic components, each with its own properties and behaviour. In a computer-based system these characteristic components are the problem world domains, interacting under the control of the software. The components are configured so that their interactions fulfil the purpose of the contrivance. The way in which the purpose is fulfilled by the components is the 'operational principle' of the contrivance: that is, the explanation of how it works. In a pendulum clock, for example, gravity acting on the weight causes a rotational force on the barrel; this force is transmitted by the gear train to the hands, causing them to turn; it is also transmitted to the escapement wheel, which rotates by one tooth for each swing of the pendulum; the rotation of the hands is therefore proportional to the number of pendulum swings; since the pendulum swings at an almost constant rate, the changing angular position of the hands indicates how much time has elapsed since they were set to a chosen reference point.

The design and the operational principle of such a machine reflect what Polanyi calls 'the logic of contrivance'. This is something different from, and not reducible to, physics or mathematics: it is the comprehension of a human purpose and the accompanying intuitive grasp of how that purpose can be achieved by the designed contrivance. It is, above all, the exercise of invention, supported by the human faculties of visualisation and embodied imagination. It is not in itself formal, although it can be applied to a formal subject matter—for instance in the conception of a mathematical theorem or the understanding of a physical process. Natasha Myers gives an account [15] of how a brilliant teacher of chemistry explained and illustrated the folding of a protein to students by physically enacting the folding process with his arms and hands. "In this process," she writes, "scientists' bodies become instruments for learning and communicating their knowledge to others." The knowledge here is, essentially the 'feel' for how the protein folding works.

5 General Laws and Specific Contrivances

The important distinction between engineering and natural science is reflected in the difference between the logic of contrivance and the laws of nature.

The scientist's ambition is to discover laws of a universal—or, at least, very general—nature. Experiments are designed to test a putative law by excluding, as rigorously as possible, all complexities that are not pertinent to the law as formulated. The generality, and hence the value, of the experimental result depends on this isolation from everything regarded as irrelevant. In a chemical experiment the apparatus must be perfectly clean and the chemicals used must be as pure as possible.

In an electrical experiment extraordinary measures may be taken to exclude stray capacitance or inductance. Ideally, only the effect of the putative law that is the subject of the experiment is to be observed: the effects of all other laws must be in some sense cancelled out or held constant.

An engineered contrivance, by contrast, is designed to work only in a restricted context implied by the engineer's mandate. The pendulum clock cannot work where there is no gravitational field of appropriate strength. It cannot work on a ship at sea, because it must maintain an upright position relative to the gravitational force and must not as a whole be subject to acceleration. It cannot work submerged in oil or water, because the resistance to pendulum motion would be too great, and the effect of the weight would be too far diminished by the upward thrust of the liquid in which it is submerged. The chosen context—albeit restricted—must then be accepted for what it is. The contrivance is intended for practical use, and must work well enough in the reality of its chosen context. The design may be frustrated by the operation of some previously unknown natural law or some phenomenon whose effects have been neglected or underestimated. These inconveniences cannot be wished away or judged irrelevant: when a failure or deficiency is revealed, the engineer must find a way to resolve the problem by improving the design. Legislating a further restriction in the context is rarely possible and never desirable. For example, when it became evident that early pendulum clocks ran faster in winter and slower in summer, it was certainly not a possible option to restrict their use to one season of the year. An engineering solution was necessary, and was found in the temperature-compensated pendulum, which maintains a constant distance between the pendulum's centre of gravity and the axis on which it swings.

Of course, the contrivance and its component parts cannot flout the laws of nature; but the laws of nature provide only the outermost layer of constraint on the contrivance's behaviour. Within this outermost layer is an inner layer of constraint due to the restricted context of use. The laws of nature would perhaps allow the engineer to predict by calculation how the clock would behave on the moon, but the prediction is pointless because the clock is not intended for operation on the moon. Within the constraints of the restricted context there is the further layer of constraint due to the engineer's design of the contrivance. The parts and their interactions are shaped by the engineer, within the boundaries left undetermined by the laws of nature as they apply within the restricted context, specifically to embody the operational principle of the contrivance.

It is the specific human purpose, the restriction of the context, and the engineer's shaping of the contrivance to take advantage of the context to fulfil the purpose, that make engineering more than science. It is a mistake to characterise engineering as an application of natural science. If it were, we would look to eminent physicists to design bridges and tunnels. We do not, because engineering is not reducible to physics. In Polanyi's words [20, p.39]:

> "Engineering and physics are two different sciences. Engineering includes the operational principles of machines and some knowledge of physics bearing on those principles. Physics and chemistry, on the other hand, include no knowledge of the operational principles of machines. Hence a complete physical and chemical topography of an object would not tell us whether it is a machine, and if so, how it works, and for what purpose. Physical and chemical

> investigations of a machine are meaningless, unless undertaken with a bearing on the previously established operational principles of the machine."

6 The Lesson of the Established Branches

Over forty years ago there was much dissatisfaction with the progress and achievements of software development, and much talk of a 'software crisis'. Although in those days the primary and central role of the physical problem world in a computer-based system had not yet become a direct focus of interest, many people nonetheless looked to the established engineering branches as a model to emulate. This was the explicit motivation of the famous NATO software engineering conferences [16,5] of 1968 and 1969:

> "In late 1967 the Study Group recommended the holding of a working conference on Software Engineering. The phrase 'Software Engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering."

Certainly, the participants were not complacent. At the 1968 meeting Dijkstra said in discussion:

> "The general admission of the existence of the software failure in this group of responsible people is the most refreshing experience I have had in a number of years, because the admission of shortcomings is the primary condition for improvement."

Yet somehow the explicit motivation of the Study Group played little part in the presentations and discussions. An invited talk by M D McIlroy [16, pp. 138–155] was perhaps a lone exception. McIlroy argued the need for a components industry that would offer catalogues of software components for input and output, trigonometric functions, or symbol tables for use in compilers; but this suggestion stimulated no broader discussion beyond the single session in which it was made.

The conference participants did not explore the practice and theory that they were invited to emulate. Had they made even a modest start in this direction, they would surely have recognised the broadest, simplest and most conspicuous lesson to be learned from the established branches. From their mere plurality—they are 'the established *branches*', not 'the established *branch*'—it can be seen at once that they are specialised. Civil engineers do not design motor cars, electrical power engineers do not design bridges, and aeronautical engineers do not design chemical plants. Our historical failure to learn and apply this lesson fully is epitomized by the persistence of the phrase 'software engineering'. Software engineering can be a single discipline only on the assumption, by analogy, that the established branches constitute a single discipline of 'tangible engineering'. On the contrary: they do not. The very successes that we hoped to emulate depend above all on their specialisation into distinct branches.

7 Specialisation Has Many Dimensions

Specialisation, in the sense that matters here, is not the concentrated focus of one individual on a single personal goal. It is the focus of an enduring community over an extended period, developing, preserving, exploiting and enlarging their shared knowledge of a particular field of technical or scientific endeavour.

Specialisations emerge and evolve in response to changing needs and opportunities, and focus on many different interlocking and cross-cutting aspects and dimensions of a field. The established branches of engineering illustrate this process in a very high degree. There are specialisations by engineering artifact—automobile, aeronautical, naval and chemical engineering; by problem world—civil and mining engineering; and by requirement—production engineering, industrial and transportation engineering. There are specialisations in theoretical foundations—control and structural engineering; in techniques for solving mathematical problems that arise in the analysis of engineering products—finite-element analysis and control-volume analysis; in engineered components for use in larger systems—electric motors, internal combustion engines, and TFT screens; in technology and materials—welding, reinforced concrete, conductive plastics; and in other dimensions too.

These specialisations do not fall into any simple hierarchical structure. They focus, in their many dimensions, on overlapping areas at every granularity, and on every concern from the most pragmatic to the most theoretical, from the kind of engineering practice that is almost a traditional craft to such theoretical topics as the advances in thermodynamics that provided the scientific basis for eliminating boiler explosions in the high-pressure steam engines of the first half of the nineteenth century.

8 The Key to Dependability Is Artifact Specialisation

In choosing whom among the established engineers we ought to emulate, there is no reason to exclude any of the many dimensions of specialisation that they exhibit; but there is a very practical reason to regard one dimension of engineering specialisation as fundamental and indispensable. This is what we may call ‘artifact specialisation’: that is, specialisation in the design and construction of artifacts of a particular class. Because a component in one engineer’s design may itself be the artifact in which another engineer specialises, and the same artifacts may appear as components in different containing artifact classes, there is a potentially complex structure of artifact specialisations. Within this structure an artifact specialisation can be identified wherever an adequately specialised engineering group exists for whom that artifact is their group product, designed for delivery to customers outside the group. The specialist artifact engineer is responsible for the completed artifact, and for the total of the value, experience and affordances it offers its users and everyone affected by it. There are no loopholes and no escape clause. This is the essence of engineering responsibility—a responsibility which is both the criterion and the stimulus of artifact specialisation.

Effective artifact specialisation is not an easy option. In its most successful forms it demands intensive and sustained research efforts by individuals and groups within the

specialist community. Walter Vincenti, in his book *What Engineers Know and How they Know It* [22], describes three remarkable examples in aeronautical engineering of the years from 1915 to 1945.

The first example addressed the problem of propeller design. Most early aircraft were driven by two-bladed propellers. The design of such a propeller is aerodynamically more complex than the design of an aircraft wing, because the propeller blades are not only moving in the direction in which the aircraft is flying but are also rotating about the propeller's axis. To address this problem, W F Durand and E P Lesley carried out detailed analytical studies and wind-tunnel tests on 150 propeller designs, working together over the years 1915–1926.

The second example concerned the problem of flush-riveting the metal skin to the frame of a fuselage or tailplane or fin. When metal skins first became practicable for aircraft they were fixed to the frames by rivets with domed heads. Soon it became apparent that the domed heads were reducing performance by causing significant aerodynamic drag: flush riveting was therefore desirable, in which the rivet head would have a countersunk profile and would not protrude above the skin surface. The skin was thin—typically, no more than 1mm thick; the necessary countersinking had to be achieved without weakening the skin or causing it to loosen under the stresses of operation. A satisfactory solution to this problem took twenty years' cooperation among several aircraft manufacturers from 1930 to 1950.

The third example has perhaps more immediately obvious counterparts in software engineering. By about 1918 pilots were beginning to articulate a strong desire for certain non-functional requirements in aircraft: they wanted aircraft with 'good flying qualities—stable, responsive, unsurprising and satisfactory to fly'. The questions arose: What did the pilots really mean by these quality requirements? What behavioural properties of the designed artifacts—the aircraft—would ensure these qualities? How could designers achieve these behavioural properties? Specialist research into these questions was conducted over the following twenty years and more. By about 1940 the main elements of the answers had been established for aircraft of the normal design of the time—that is, aircraft with lateral symmetry, a single wing with a straight leading edge, and a horizontal tailplane and vertical fin at the rear. The answers were inevitably very technical. They applied to the established standard design; they were expressed in terms of that design; and finding them took the dedicated specialised effort of designers, pilots and instrument engineers over a quarter of a century.

9 Normal and Radical Design

The fruit and direct expression of each artifact specialisation is the associated 'normal design': that is, the engineering design practice that is standard in the specialisation, and the standard design products that it creates. Vincenti characterises normal design like this [22]:

> "[...] the engineer knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task."

He illustrates normal design by the practice of aero engine design in the period before 1945:

> "A designer of a normal aircraft engine prior to the turbojet, for example, took it for granted that the engine should be piston-driven by a gasoline-fueled, four-stroke, internal-combustion cycle. The arrangement of cylinders for a high-powered engine would also be taken as given (radial if air-cooled and in linear banks if liquid-cooled). So also would other, less obvious, features (e.g., tappet as against, say, sleeve valves). The designer was familiar with engines of this sort and knew they had a long tradition of success. The design problem—often highly demanding within its limits—was one of improvement in the direction of decreased weight and fuel consumption or increased power output or both."

In *radical design*, by contrast,

> "[…] how the device should be arranged or even how it works is largely unknown. The designer has never seen such a device before and has no presumption of success. The problem is to design something that will function well enough to warrant further development."

No design is ever completely and utterly radical in every respect, because the designer's previous knowledge and experience inevitably bring something potentially relevant and useful to the task. Karl Benz, for example, had worked successfully on the design of a gasoline-fuelled engine before he applied for a patent [2] for his first complete automobile design. Details of Benz's design and photographs of exact replicas (two series of replicas were manufactured in the late twentieth century) are easily available on the web [1]. The car had three wheels, each with radial metal spokes and solid rubber tyres. The driver steered by a crank handle that turned the vertically pivoted front wheel through a rack-and-pinion mechanism. The car was powered by a single-cylinder four-stroke engine with an open crankcase, mounted behind the bench seat between the rear wheels, and positioned with the crankshaft vertical. A large open flywheel was mounted on the crankshaft, and to start the engine the flywheel was rotated by hand. Power was transmitted to the wheels through bevel gears, to a belt drive that acted also as a clutch, a differential, and finally by a chain drive to each rear wheel. Braking was provided by a handbrake acting on an intermediate shaft.

This design incorporated many innovations and was undoubtedly a work of genius. As a practicable engineering artifact it satisfied Vincenti's criterion—it worked well enough to warrant further development—but it had many defects. The most notable failure was in the design of the steering. The single front wheel made the ride unstable on any but the smoothest surface, and the very small radius of the crank by which the vehicle was steered gave the driver too little control. Control could be improved by a longer crank handle, but the single front wheel was a major design defect. It is thought that Benz adopted the single front wheel because he was unable to solve the problem of steering two front wheels. To ensure that the trajectory of each front wheel is tangential to the path of travel, the inner wheel's turning angle must be greater than the outer's. The arrangement needed to achieve this—now commonly known as 'Ackermann steering geometry'—had been patented some seventy years earlier by a

builder of horse-drawn carriages and also was used ten years earlier in a steam-powered vehicle designed in France, but neither Benz nor his contemporary Gottlieb Daimler was aware of it. Five years later Benz rediscovered the principle, and even succeeded in obtaining a patent in 1891. His 1893 car, the Victoria, had four wheels and incorporated Ackermann steering.

10 Artifact Specialisations in Software Engineering

Software engineering certainly has many visible specialisations. There are many active specialisations in theoretical areas that may be regarded as belonging to computer science but have direct application to the engineering of software-based systems: for example, concurrency and complexity theory. There is a profusion of specialisations in relevant tools and technologies: for example, software verification, model-checking, formal specification languages, and programming languages for developing web applications.

There are artifact specialisations too, but chiefly for artifacts whose problem world, while not being purely abstract, remains comfortingly remote from the complications and uncertainties of the physical and human world. Fred Brooks characterised this class of artifact neatly in a comment [4, pp. 56–57] on the open source movement:

> "The conspicuous success of the bazaar process in the Linux community seems to me to derive directly from the fact that the builders are also the users. Their requirements flow from themselves and their work. Their desiderata, criteria and taste come unbidden from their own experience. The whole requirements determination is implicit, hence finessed. I strongly doubt if Open source works as well when the builders are not themselves users and have only secondhand knowledge of the users' needs."

The admirable principle "eat your own dogfood" applies well when the developers are also the customers, or can easily play the customer's role with full conviction. So there are effective specialisations working on compilers, file systems, relational database management systems, SAT solvers and even spellcheckers. Use of these artifacts falls well within the imaginative purview of their developers, and the buzzing blooming confusion of the physical world is safely out of sight.

The same can not be said of radiotherapy or automotive systems, or of systems to control a nuclear power station or an electricity grid. It is not yet clear to what extent these systems, and many other computer-based systems that are less critical but still important, have begin to benefit from the intensive specialisation that has marked the established engineering branches over their long histories. Even when a specialisation itself has become established the evolution of satisfactory normal design is likely to take many years. Karl Benz's invention had given rise to an international industry by about 1905; but it was not until the 1920s that automobiles could be said to have become the object of normal design.

11 Artifact Specialisation Needs Visible Exemplars

There are many cultural, social and organisational obstacles to the emergence and development of artifact specialisations in software engineering. One obstacle is what may be called a preference for the general and a corresponding impatience with the particular. In journals and conferences descriptions of specific real systems are rarely found. At most a confected case study may be presented as a sketched example to support a general thesis. We seem to assume that there is little worth learning that can be learned from a detailed study of one system.

This disdain for the specific militates strongly against the growth of effective artifact specialisations. In the established engineering branches practising engineers learn from specific real artifacts about the established normal design, their properties and their failures. For example, the Earthquake Engineering Research Center of UC Berkeley maintains a large library [8] of slides showing exemplars of real civil engineering structures such as bridges and large buildings of many kinds. The collection is arranged primarily by artifact class, and serves as a teaching resource for undergraduate and graduate courses. Students learn not only by acquiring knowledge of theory, but also by informed examination of specific real engineering examples: each example has its place in a rich taxonomy, and its own particular lessons to teach.

Failures are no less—perhaps even more—important than successes. Every engineering undergraduate is shown the famous amateur movie that captured the collapse [9] of the Tacoma Narrows Bridge in 1940. The designer, Leonid Moisseiff, had given too little consideration to the aerodynamic effects of wind on the bridge's narrow and shallow roadway, and especially to vertical deflections. In a wind of only 40 mph, vertical oscillations of the roadway built up to a magnitude that completely destroyed the bridge. Designers of suspension bridges learned the lesson, and have recognised their obligation to preserve it and hand it on to their successors. It is a very specific lesson. Moisseiff's mistake will not be repeated.

In software engineering we can be less confident that we will not repeat the mistakes we discover in our designs. Without artifact specialisation there may be no structure within which a sufficiently specific lesson can be learned. Some years after the Therac-25 experiences an excellent investigative paper [13] was published in IEEE Computer and appeared in an improved form as an appendix to [14]. The researchers studied all the available evidence, and their paper identified certain specific software and system errors. But the absence of a normal design for the software of a radiotherapy machine was clearly evidenced by the lack of nomenclature for the software components. The electro-mechanical equipment parts were given their standard names in the diagrams that appeared in the paper: the turntable, the flattener, the mirror, the electron scan magnet, and so on. But the software parts had no standard names, and were referred to, for example, as "tasks and subroutines in the code blamed for the Tyler accidents", individual subroutines being given names taken from the program code. Inevitably, the recommendations offered at the end of the paper concentrated chiefly on general points about the development process that had been used: they focused on documentation, software specifications, avoidance of dangerous coding practices, audit trails, testing, formal analysis and similar concerns. No specific recommendations could be made about the

software artifact itself, because there was no standard normal design to which such recommendations could refer. The errors of the Therac-25 software engineers were, regrettably, repeated twenty five years later.

12 The Challenge of Software System Structures

Almost every realistic computer-based system is complex in many planes. This complexity is a major obstacle to the design of a system, and to communicating and explaining its structure. It is also, therefore, a major obstacle to the evolution of normal designs.

More properly, we should speak of many structures, not just of one. The system provides many features, each one an identifiable unit of functionality that can be called into play according to circumstances and user demands. Each feature can be regarded as realised by an associated contrivance. The components of the contrivance are problem domains and parts or projections of the system's software, arranged in a suitably designed configuration within which they interact to fulfil the functionality of the feature.

The features will inevitably interact: they may share problem domains and software resources; several may be active in combination at any time; and their requirements may conflict. These interactions and combinations will itself induce a distinct structure, demanding careful study and analysis both in terms of the requirements they are expected to satisfy and in terms of the implementation. This *feature interaction problem* [23] came to prominence in the telephone systems of the late twentieth century, where interactions of call processing features such as call forwarding and call barring were seen to produce surprising and sometimes potentially harmful results. The structure of these interactions might be captured, for example, in graphs of the kind used in Distributed Feature Composition [11]. The feature interaction problem hugely increased the costs and difficulties of building telephone system software. It soon became apparent that the same problem was inherent in complex systems of every kind.

The system as a whole, regarded as a single contrivance, will have several contexts of operation. For example, a system to control lifts in a building containing offices, shops and residential apartments must take account of the different demand patterns imposed from hour to hour and from day to day by this usage. There will also be extraordinary operational contexts such as emergency operation under control of the fire department, operation under test control of a representative of the licensing authority, and operation during periodic maintenance of the equipment. If an equipment fault is detected the system must take action to mitigate the fault if possible—for example, by isolating the faulty subset of the equipment or by executing emergency procedures to ensure the safety of any users currently travelling in the lift cars. These contexts are not necessarily disjoint: for example, a fault may be detected during testing, and the safety of the test engineer must be ensured.

The properties and behaviours of the individual problem world domains—the lift shafts, the floors, the doors, the lift cars, the users, the hoist motors—must be analysed in the development process. The design of the contrivance realising each

feature must take account of the relevant properties and behaviours of those problem domains that figure as components in its design. It may be necessary for the executed software to embody software objects that model the problem domains. A domain will then have a model for each feature in which it participates, and these models must be reconciled and in some way combined.

The implemented software itself will have some kind of modular structure, its parts interacting by the mechanisms provided by the execution infrastructure and the features of the programming languages used. The extreme malleability of software, which is not shared by physical artifacts, allows the implementation structure to differ radically from the other structures of the system. The relationships among these structures may then be mediated by transformations.

All of these structures must be somehow captured, designed, analysed, reconciled and combined. The task can certainly not be performed successfully by simplistic generalised techniques such as top-down decomposition or refinement. Nor can it be evaded by the kind of reductionist approach that fragments the system into a large collection of elements—for example, events or objects or one-sentence requirements. In general, this is an unsolved problem. In a normal design task a satisfactory solution has been evolved over a long period and is adopted by the specialist practitioners. Where no normal design is available, and major parts and aspects of the design task are unavoidably radical, the full weight of the difficulty bears down on the designer. This difficulty is a salient feature of the design of many computer-based systems.

13 Forgetting the Importance of Structure

In an earlier era, when computers were slower, programs were smaller and computer-based systems had not yet begun to occupy the centre of attention in the software world, program structure was recognised as an important topic and object of study. The discipline of structured programming, as advocated [6] by Dijkstra in the famous letter, was explicitly motivated by the belief that software developers should be able to understand the programs they were creating. The key benefit of the structure lay in a much clearer relationship between the static program text and the dynamic computation. A structured program provides a useful coordinate system for understanding the progress of the computation: the coordinates are the text pointer and the execution counters for any loops within which each point in the text is nested. Dijkstra wrote:

> "Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process."

A structured program places the successive values of each state component in a clearly defined context that maps in a simple way to the evolving state of the whole computation. The program structure tree shows at each level how each lexical component—elementary statement, loop, if-else, or concatenation—is positioned within the text. If the lexical component can be executed more than once, the execution counter for each enclosing loop shows how its executions are positioned

within the whole computation. The programmer's understanding of each part of the program is placed in a structure of nested contexts that reaches all the way to the root of the structure tree.

Other structural notions beyond the control structure discipline advocated for structured programming—for example, layered abstractions and Naur's discipline [17] of programming by action clusters—were explored, always with the intention of simplifying and improving program development by supporting human understanding. The primary need was for intelligible structure. Rigorous or formal proof of correctness would then be a feasible by-product, the structure of the proof being guided by the structure of the designed program.

Work on the formal aspects of this plan proved brilliantly successful during the following years. Unfortunately, however, it was so successful that the emphasis on human understanding gradually faded, and many of the most creative researchers simply lost interest in the human aspects. It began to seem more attractive to allow the formal proof techniques to drive the program structuring. By 1982 Naur was complaining [18]:

> "Emphasis on formalization is shown to have harmful effects on program development, such as neglect of informal precision and simple formalizations. A style of specification using formalizations only to enhance intuitive understandability is recommended."

For small and simple programs it was feasible to allow a stepwise proof development to dictate the program structure. Typically, the purpose of the program was to establish a result that could easily be specified tersely and formally: the proof was then required to establish this result from the given precondition. But Naur's complaint was justified. It is surely a sound principle that human understanding and intuition must be the foundation of software engineering, not a primitive scaffolding that can be discarded once some formal apparatus has been put in place.

The success of the formalist enterprise has led to a seriously diminished interest in structural questions and concerns, and in the role of human understanding, just when they are becoming more important. The constantly increasing complexity of computer-based systems cannot be properly addressed by purely formal means. Forty five years ago, the apparently intractable problem of designing and understanding program flowcharts yielded to the introduction of structure in the service of human understanding. Managing and mastering the complexity of the systems being built now and in the future demands human understanding: formal techniques are essential, but they must be deployed within humanly intelligible structures.

References

1. http://www.youtube.com/watch?v=9MjvAqF9Tgo (last accessed 3rd July 2010).
2. http://www.unesco-ci.org/photos/showphoto.php/photo/5993/title/benz-patent-of-1886/cat/1009 (last accessed 4th July 2010).
3. Bogdanich, W.: The Radiation Boom. New York Times (January 23 and January 26, 2010)

4. Brooks, F.P. Jr: The Design of Design: Essays from a Computer Scientist. Addison-Wesley (2010)
5. Buxton, J.N., Randell, B. (Eds.): Software Engineering Techniques. Report on a conference sponsored by the NATO SCIENCE COMMITTEE, October 1969; NATO (April 1970)
6. Dijkstra, E.W.: A Case Against the Go To Statement, EWD 215, published as a letter to the Editor (Go To Statement Considered Harmful): Communications of the ACM 11(3), pp. 147-148 (March 1968)
7. Dijkstra, E.W.: Reply to comments on On the Cruelty of Really Teaching Computer Science. Communications of the ACM 32(12), pp. 1398-1404 (December 1989)
8. Godden, W.G.: http://nisee.berkeley.edu/godden/godden_intro.html (last accessed 25 June 2010)
9. Holloway, C.M.: From Bridges and Rockets, Lessons for Software Systems. In: Proceedings of the 17th International System Safety Conference, pp. 598-607 (August 1999)
10. Jackson, D., Thomas, M., Millett L.I. (Eds.): Software for Dependable Systems: Sufficient Evidence? Report of a Committee on Certifiably Dependable Software Systems; The National Academies Press, Washington DC, USA, page 40 (2007)
11. Jackson, M., Zave, P.: Distributed Feature Composition: A Virtual Architecture For Telecommunications Services. IEEE Transactions on Software Engineering 24(10), pp. 831-847, Special Issue on Feature Interaction (October 1998)
12. Krebs, B.: Cyber Incident Blamed for Nuclear Power Plant Shutdown; Washington Post (June 5, 2008)
13. Leveson, N.G., Turner, C.S.: An Investigation of the Therac-25 Accidents. IEEE Computer 26(7), pp. 18-41 (July 1993)
14. Leveson, N.G.: Safeware: System Safety and Computers. Addison-Wesley (1995)
15. Myers, N.: Performing the Protein Fold. In: Sherry Turkle, Simulation and Its Discontents. MIT Press, pp. 173-4 (2009)
16. Naur, P., Randell, B. (Eds.): Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, October 1968. NATO (January 1969)
17. Naur, P.: Programming by Action Clusters. BIT 9(3), pp. 245-258 (September 1969); republished in: Naur, P.: Computing, A Human Activity. ACM Press, pp. 335-342, 1992
18. Naur, P.: Formalization in Program Development; BIT 22(4), pp. 437-452 (December 1982); republished in: Naur, P.: Computing, A Human Activity. ACM Press, pp. 433-449, 1992
19. Polanyi, M.: Personal Knowledge: Towards a Post-Critical Philosophy. Routledge and Kegan Paul, London, 1958, and University of Chicago Press (1974)
20. Polanyi, M.: The Tacit Dimension. University of Chicago Press (1966). with foreword by Amartya Sen, 2009
21. http://catless.ncl.ac.uk/risks/ (last accessed 1st July 2010).
22. Vincenti, W.G.: What Engineers Know and How They Know It: Analytical Studies from Aeronautical History. The Johns Hopkins University Press, Baltimore (1993)
23. Zave, P.: http://www2.research.att.com/~pamela/fi.html; FAQ Sheet on Feature Interaction. AT&T (2004)

Tools and Behavioral Abstraction: A Direction for Software Engineering

K. Rustan M. Leino

Microsoft Research, Redmond, WA, USA
leino@microsoft.com

Capsule Summary. As in other engineering professions, software engineers rely on tools. Such tools can analyze program texts and design specifications more automatically and in more detail than ever before. While many tools today are applied to find new defects in old code, I predict that more software-engineering tools of the future will be available to software authors at the time of authoring. If such analysis tools can be made to be fast enough and easy enough to use, they can help software engineers better produce and evolve programs.
A programming language shapes how software engineers approach problems. Yet the abstraction level of many popular languages today is not much higher than that of C programs several decades ago. Moreover, the abstraction level is the same throughout the program text, leaving no room for *behavioral abstraction* where the design of a program is divided up into stages that gradually introduce more details. A stronger arsenal of analysis tools can enable languages and development environments to give good support for behavioral abstraction.

0 Introduction

The science and practice of software engineering have made great strides in the few decades since its inception. For example, we have seen the rise of structured programming, we have come to appreciate types (enforced either statically or dynamically), we have developed models for program semantics that underlie reasoning about programs, we have recognized the role of (unit, system, regression, white-box, ...) testing, we are starting to understand how to collect metrics that help in setting software-development schedules, and, having available far more CPU cycles than would have been easily imaginable in 1968, we have produced tools that assist in all these tasks.

Nevertheless, software engineering remains difficult and expensive.

What is it all about? Software engineering produces software—if we had no interest in software, the activities that make up software engineering would have no purpose. We have a number of desiderata for this engineering task. We want to develop software...

– **with the right features.** A software project is successful only if the final software does what its users need it to do. The requirements for a piece of software can be difficult to determine, the assumptions made by users and the software team may be in conflict with each other, and that which seems important when prototyping may end up being different from what is important during deployment of the final software.

S. Nanz (ed.), *The Future of Software Engineering*,
DOI 10.1007/978-3-642-15187-3_7,

- **that is easy to use.** At times, we have all been frustrated at desktop software where we found menus to be unnecessarily cumbersome or non-standard, and at web software where the number of clicks required for a purchase seems ridiculously high. Software in a car must allow operation while the driver stays on task.
- **that is hard to misuse,** both accidentally and maliciously. Letting an operator open up the landing gear of an airplane too far off the ground may not be what we want of our embedded software. And we expect banking software to apply judicious security measures to prevent various forms of privacy breaches.
- **can be developed effectively,** on schedule and free of defects. Ideally, we want all software to have zero defects (like crashes, deadlocks, or incorrect functional behavior). But reducing defects comes at a price, so market forces and engineering concerns may bring about compromises. Whatever defect rates can be tolerated, we would like the software engineering process to get there as easily and confidently as possible.
- **can be evolved,** to add or remove features, to adapt to new environments, to fix defects, and to preserve and transfer knowledge between developers. Successful software lives far beyond its first version, an evolution in which feature sets and development teams change. As the software is changed, one needs to understand what modifications are necessary and also which modifications are possible.

The future of software engineering holds improvements in these and related areas. In this essay, I will focus on issues concerning the software artifacts themselves, paying much less attention to the important issues of determining requirements and of usability of the software itself.

1 Composing Programs

The phrase "composing programs" has two senses. One sense is that of authoring programs, like a musician composes music. Another sense is that of combining program elements to form a whole program. When we create a piece of software, we take part in activities that comprise both senses of "composing programs": we both write new code, designing new data structures and algorithms, and combine existing or planned program elements, calling libraries and reusing types of data structures.[0] Both activities wrestle with complexity. Complexity arises in the algorithms used, in the data structures used, and in the interaction between features. The major tool for keeping complexity at bay is *abstraction*.

Common Forms of Abstraction. When authoring new program elements, abstraction is used to group related functionality to make larger building blocks. Standard programming languages offer several facilities for abstraction. *Procedural abstraction* provides the ability to extend the primitive operations of the language with user-defined, compound operations. These can be named and parameterized, and they separate uses of the

[0] Encompassing both senses, the IFIP Working Group 2.3 on *Programming Methodology* states as its aim, "To increase programmers' ability to compose programs".

operation from its implementation. *Data abstraction* provides the ability to extend the primitive data types of the language with user-defined, compound data types. These can be named and sometimes parameterized, and they can separate uses of the data from the actual representation.

For example, one part of a program can create and apply operations to a file stream object without having to understand how many underlying buffers, file handles, and position markers are part of the actual data representation or to understand the operations of the protocol used by the disk to fetch the contents of a particular sector.

When combining program elements, abstraction is needed to let a program element declare indifference about certain aspects of its uses. The main facility for this is *parameterization*. This is as important for the provider of a program element as it is for clients of the program element. Sometimes, there are restrictions on how a program element can be parameterized. Such restriction can be declared, for example, as type constraints or precondition contracts.

For example, a sorting procedure can be parameterized to work for any data type, as long as the objects of that data type have some means of being compared. This can be declared as a type constraint, requiring the sorting procedure's type parameter to be a type that implements a $Compare$ method. Alternatively, the sorting procedure can be parameterized with a comparison procedure, and a precondition of the sorting procedure can require the comparison to be transitive.

Abstraction by Occlusion. These and similar abstraction facilities go a long way to organizing a program into understandable and manageable pieces. But not far enough. There is one major aspect of abstraction that is conspicuously underdeveloped in popular languages. The facilities for abstraction I mentioned above provide a way to occlude parts of a program, revealing the insides of those parts only in certain scopes. For example, the callers of a procedure see only the procedure signature, not its implementation, and the fields of a class may be visible only to the implementation of the class itself, not to clients of the class. We may think of this as *abstraction by occlusion*. What is lacking in this kind of abstraction is that it only gives us two options, either we have to try to understand a particular part of the program by digging into its details or we don't get any information to help us understand it. Stated differently, a view of a program is a subset of the code that is ultimately executed.

For example, consider a type that represents a graph of vertices and edges, and suppose you want to evolve the functionality of this type, perhaps to fix a defect or add a feature. When you look at the implementation, you immediately get immersed with the details of how edges are represented (perhaps there is a linked list of vertex pairs, perhaps each vertex stores an array of its edge-ordering successors, or perhaps both of these representations are used, redundantly) and how results of recent queries on the graph may be cached. This may be more details than you need to see. Furthermore, suppose you change the explicit stack used by an iterative depth-first graph traversal from using a vertex sequence to reusing reversed pointers in the graph à la Schorr-Waite. You will then remove the previous vertex sequence, which means that the next person to look at this code will have to understand the complicated pointer swiveling rather than the easier-to-grasp stack of vertices. Finally, the fact that you would remove the

previous vertex-sequence implementation is unfortunate, because it would have been nice to have kept it around as a reference implementation.

Behavioral Abstraction. Abstraction by occlusion does not provide a good story for understanding more abstractly what the code is supposed to do, or even giving a more abstract view of how the code does what it does. If you were to explain a piece of software to a colleague, you are not likely to explain it line by line. Rather, you would first explain roughly what it does and then go into more and more details. Let me call this *behavioral abstraction*.

While behavioral abstraction is mostly missing from popular languages, it does exist in one important form: *contracts*, and especially procedure postcondition contracts, which are found in some languages and as mark-ups of other languages [17, 16, 6, 7, 5]. A postcondition contract abstracts over behavior by giving an expression that relates the possible pre- and post-states of the procedure. To give a standard example, the postcondition of a sorting procedure may say that its output is a permutation of the input that arranges the elements in ascending order. Such a postcondition is easier for a human to understand than trying to peer into the implementation to figure out what its loops or recursive calls do. It also opens the possibility for verification tools to compare the implementation with the intended behavior of the procedure as specified by the postcondition.

A weaker form of behavioral abstraction is found in software engineering outside programming languages, namely in test suites. To the extent that a test suite can be seen as a set of use cases, it does provide a form of behavioral abstraction that, like contracts, shows some of the intent of the program design.

The behavioral abstraction provided by contracts and use-case test suites provide one layer of description above the actual code. But one can imagine applying several layers of behavioral abstraction.

Stepwise Refinement. The development of many programs today goes in one fell swoop from sketchy ideas of what the program is supposed to do to low-level code that implements the ideas. This is not always wrong. Some programmers have tremendous insights into how, for example, low-level network protocols for distributed applications ought to behave. In such cases, one may need tools for verification or property discovery to ensure that the program has the desired properties. But, upon reflection, this process of writing code and *then* coming up with properties that the code should have and trying to ascertain that the code does indeed have those properties seems terribly twisted. Why wouldn't we want to *start* by writing down the salient properties of the software to be constructed, and then in stages add more detail, each maintaining the properties described in previous stages, until the program is complete? This process is called *stepwise refinement*, an old idea pioneered by Dijkstra [8] and Wirth [26].

Well, really, why not? There are several barriers to applying this technique in practice. Traditionally, stepwise refinement starts with a high-level specification. Writing high-level specifications is hard, so how do we know they are correct? While programming is also hard, a program has the advantage that it can be executed. This allows us to subject it to testing, simulation, or other use cases, which can be more immediately

satisfying and can, in some development organizations, provide management with a facade of progress milestones.

If we are to use layers of behavioral abstraction, then we must ensure that any stage of the program, not just the final program, can be the target of modeling, simulation, verification, and application of use cases.

Another problem with stepwise refinement—or, a misconception, really—is that the high-level specification encompasses all desired properties of the final program. In reality, the specification is introduced gradually, just like any other part of the design. For example, the introduction of an efficient data structure or a cache may take place at a stage in the middle of the refinement process. By splitting the specification in this way, each piece of it may be easier to get right.

Finally, a problem to overcome with stepwise refinement is that so much of the programming culture today is centered around final code. It will take some training to start writing higher-level properties instead, and engineers will need to learn when it is worthwhile to introduce another stage versus when it is better to combine some design decisions into one stage. There is good reason to believe that software engineers will develop that learning, because engineers have learned similar trade-offs in today's practices; for example, the trade-off of when it is a good idea to introduce a new procedure to perform some series of computational steps versus when it is better to just do the steps wherever they are needed.

Let's consider a vision for what a development system that embraces behavioral abstraction may look like.

2 A Development Environment for Behavioral Abstraction

First and foremost, a development environment is a tool set that allows engineers to *express and understand designs*. The tools and languages we use guide our thinking when developing a program and are key to human comprehension when reading code. By supporting behavioral abstraction, the environment should allow engineers to describe the system at different levels of abstraction. Here are some of the characteristics of such a development environment.

Descriptions at Multiple Stages. The environment permits and keeps track of increasingly detailed descriptions of the system, each a different *stage*. A stage serves as a behavioral abstraction of subsequent stages.

Ceaseless Analysis. At each stage, the environment analyzes the given program. For example, it may generate tests, run a test suite, simulate the description, verify the description to have certain well-formedness properties (*e.g.*, that it parses correctly, type checks, and adheres to the usage rules (preconditions) of the operations it uses), infer properties about the description, and verify that the stage maintains the properties described in previous stages.

To the extent possible, trouble spots found by such analyses only give rise to warnings, so as not to prevent further analyses from running.

There is no reason for the analyses to be idle, ever. Instead, the environment ceaselessly runs the analyses in the background, noting results as they become available.

Multiple Forms of Descriptions. The descriptions at one stage can take many different forms. For example, they may spell out a use case, they may prescribe pre- and postcondition contracts, they may give an abstract program segment, they may introduce variables and associated invariants, or they may supply what we today think of as an ordinary piece of program text written in a programming language. Many of these descriptions (like test cases and weak postconditions) are incomplete, homing in on what is important in the stage and leaving properties irrelevant to the stage unspecified.

In some cases, it may be natural to switch languages when going into a new stage. For example, a stage using JavaScript code may facilitate the orchestration of actions described in more detailed stages as C# code for Silverlight [18]. Other examples of orchestration languages include Orc [20] and Axum [19].

Note, since one form of description is code in a programming language, one could develop software using this system in the same way that software is developed today. This is a feature, because it would allow engineers to try out the capabilities of the new system gradually. One gets a similar effect with dynamically checked contracts: if a notation for them is available, programmers can start taking advantage of them gradually and will get instant benefits (namely, run-time checks) for any contract added. In contrast, if a machine readable format for contracts is not available, contracts that appear in the head of an engineer while writing or studying the code turn into lost opportunities for tools to help along.

Change of Representation. It is important that a stage can refine the behavior of a previous stage not just algorithmically—to make an incomplete description more precise—but also by changing the data representation. For example, the description at one stage may conveniently use mathematical types like sets and sequences, whereas a subsequent stage may decide to represent these by a list or a tree, and an even later stage may want to add some redundant representations, like caches, to make the program more efficient.

By describing an optimization as a separate stage, the overall design can still be understood at previous stages without the code clutter that many good optimizations often bring about.

This kind of transformation is known as *data refinement* and has been studied extensively (*e.g.*, [12, 3, 21, 22, 9]).

Executable Code. The development environment allows designs to be described down to the level of the executable code of the final product. That is, the development environment is not just for modeling. However, the code need not all be produced by traditional compilers; parts of it may be synthesized from higher-level descriptions. Similar ideas have been explored in the language SETL [23] and in correct-by-construction synthesis frameworks [24].

Automation. For the most part, we think of program analysis as an automatic process. Sometimes, however, an analysis may need manually supplied hints to improve the analysis or to keep it from getting stuck. For example, a static type checker may need a dynamic type cast, a type inference engine may need type annotations for recursive procedures, and a proof assistant may need a lemma or a proof-tactic command. To provide a seamless integration of such analysis tools, the hints are best supplied using concepts at the level of the description at hand. For example, an assert statement in a program text can serve as a lemma and may, in that context, be more readily understood by an engineer than an indication that a certain proof-tactic command needs to be invoked at some point in the proof.

Room for Informality. The development environment allows formal verification of properties in and between stages. Indeed, such verification may be easier than in a monolithic program with specifications, because behavioral abstraction naturally breaks up the verification tasks in smaller pieces. However, it is important not to prevent programs from being written, simulated, or executed just because they cannot be formally verified at the time. Instead, an engineer has the option to accept responsibility for the correctness of certain conditions.

3 Challenges

A number of challenges lie ahead before a development environment built around behavioral abstraction can become a reality. Here are some of them.

User Interface. A behavioral-abstraction environment could benefit from a different user interface than the file-based integrated development environments of today. Conceptually, a stage contains only the differences in the program since the previous stage. Sometimes, one may want to see only these differences; at other times, one may want to see the combined effect of a number of stages.

Early Simulation. A stage is more useful if something can be done with it. The closer something is to executable code, the easier it is to imagine how it may be simulated. How should one simulate the program in its most abstract and most incomplete stages?

Tools in the spirit of Alloy [13] or Formula [14] would be useful here.

Prioritizing Analyses. To run analyses ceaselessly is harder than one might first imagine. Many analyses are designed with various compromises in mind, terminate quickly, and then have nothing more to do. For example, a type checker may be designed to report an error when types don't match up; but instead of being idle after warning about such a condition, it could continue being useful by searching for program snippets that supply possible type conversions. If an analysis could use boundless amounts of CPU time, the development environment must use some measures of priority. For example, if the engineer just modified the implementation of a procedure, it would seem a good

idea to give priority to analyses that can tell the engineer something useful about the new implementation. This may require recognition of differences in the abstract syntax tree (*e.g.*, to re-analyze only those code paths that were changed), keeping track of dependencies (*e.g.*, the call graph), and maybe even using a program verifier's proof to determine detailed dependencies (*e.g.*, to figure out that one procedure relies only on some specific portions of other specifications).

Allowing Informality. Ideally, we want a program to be all correct in the end. But before then, engineers may be able to be more productive if they can continue simulating or testing their program even in the presence of some omissions or errors. How to go about doing this is a challenge. Perhaps we can draw inspiration from dynamically typed languages that try hard to give some interpretation to code at run time, even if a static type checker would have had a hard time with the program. Also, what run-time checks could be used to make up for failed or incomplete proofs?

Refinements into Dynamically Allocated State. Many modern applications require, or at least benefit from, use of dynamically allocated state. Yet, this issue has not received nearly the same amount of attention in data refinement as it has in the area of program verification [11].

Supporting Program Evolution. Getting a new program together is a big feat. But even more time will be spent on the program as is evolves into subsequent versions. To determine if a development system really helps in program evolution, one needs to undertake some long-running deployments.

4 Related Work and Acknowledgments

Many people have had ideas in this space and much work has been done on various components. I cannot do them all justice here, but in addition to what I have already mentioned above, I will mention a number of efforts and people who have been most influential in my own thinking.

Built on the Event-B formalism [1], the Rodin system [2] is perhaps the closest of existing systems to what I have sketched. With its pluses and minuses, it sports an integrated development environment, uses a sequence of files to record stages of the design, mostly unifies descriptions into one form ("events"), lacks direct support for dynamic memory allocation, does not routinely output executable code, and requires a noticeable amount of prover interaction.

Other important systems that let designs be described at various levels of abstraction are B [0] and VDM [15]. The Play-in, Play-out technique [10] simulates systems from given use cases.

Bertrand Meyer pioneered the inclusion of behavioral descriptions (contracts) of program elements as compiled constructs in a modern programming language [17].

When using verification tools today, one of the focus activities is finding specifications that describe what the code does. It would seem a better use of our time to shift

that focus to finding code that satisfy specifications. In some cases, programs can even be automatically synthesized from specifications, as has been investigated, for example, by Doug Smith and colleagues [25, 24].

This essay has also benefited from ideas by and discussions with Jean-Raymond Abrial, Mike Barnett, Michael Butler, Mike Ernst, Leslie Lamport, Michał Moskal, and Wolfram Schulte, as well as Mark Utting, who formulated some ideas around a collection of descriptions on which tools operate to produce efficient code and report potential problems. Kuat Yessenov worked as my research intern on language and verification support for refinement.

I have been personally influenced by Ralph Back, whose ground-breaking and inspiring work gave stepwise refinement mathematical rigor [3, 4]. Work by Shaz Qadeer and Serdar Tasiran hammered home to me how much simpler invariants can be if they are written for an abstract program instead of for the final code. A number of people have reminded me that some descriptions are most easily formulated as code; among them, Wolfram Schulte, Ras Bodik, and Gerwin Klein. Cliff Jones has repeatedly pointed out the importance of requirements and the high cost incurred by getting them wrong.

Bart Jacobs and Rosemary Monahan provided comments on a draft of this essay.

5 Conclusion

Software engineering uses lots of tools. To take a bigger step in improving software engineering, we need not just to improve each tool, but to combine tools into usable development environments. The time is ripe for that now. As a vision for how tools and languages can be combined in the future of software engineering, I discussed in this essay a development environment built around behavioral abstraction, where programs are divided up not just into modules, types, and procedures, but also according to the level of abstraction at which they describe the program under development.

References

0. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An open toolset for modelling and reasoning in Event-B. International Journal on Software Tools for Technology Transfer (April 2010)
3. Back, R.-J.: On the Correctness of Refinement Steps in Program Development. PhD thesis, University of Helsinki (1978) Report A-1978-4.
4. Back, R.-J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Graduate Texts in Computer Science. Springer-Verlag (1998)
5. Barnett, M., Fähndrich, M., Logozzo, F.: Embedded contract languages. In ACM SAC - OOPS. ACM (March 2010)

6. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices, volume 3362 of Lecture Notes in Computer Science, pages 49–69. Springer (2005)
7. Baudin, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, version 1.4 (2009) http://frama-c.com/.
8. Dijkstra, E.W.: A constructive approach to the problem of program correctness. BIT, 8:174–186 (1968)
9. Gries, D., Volpano, D.: The transform — a new language construct. Structured Programming, 11(1):1–10 (1990)
10. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart Play-out of behavioral requirements. In Mark Aagaard and John W. O'Leary, editors, Formal Methods in Computer-Aided Design, 4th International Conference, FMCAD 2002, volume 2517 of Lecture Notes in Computer Science, pages 378–398. Springer (November 2002)
11. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.: Behavioral interface specification languages. Technical Report CS-TR-09-011, University of Central Florida, School of EECS (2009)
12. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica, 1(4):271–281 (1972)
13. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
14. Jackson, E.K., Seifert, D., Dahlweid, M., Santen, T., Bjørner, D., Schulte, W.: Specifying and composing non-functional requirements in model-based development. In Alexandre Bergel and Johan Fabry, editors, Proceedings of the 8th International Conference on Software Composition, volume 5634 of Lecture Notes in Computer Science, pages 72–89. Springer (July 2009)
15. Jones, C.B.: Systematic Software Development Using VDM. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition (1990)
16. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes, 31(3):1–38, (March 2006)
17. Meyer, B.: Object-oriented Software Construction. Series in Computer Science. Prentice-Hall International (1988)
18. Microsoft: Silverlight. http://www.microsoft.com/silverlight/.
19. Microsoft: Axum. http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx (2010)
20. Misra, J., Cook, W.R.: Computation orchestration: A basis for wide-area computing. Software and Systems Modeling, 6(1):83–110, (March 2007)
21. Morgan, C.: Programming from Specifications. Series in Computer Science. Prentice-Hall International (1990)
22. Morris, J.M.: A theoretical basis for stepwise refinement and the programming calculus. Science of Computer Programming, 9(3):287–306, (December 1987)
23. Schwartz, J.T., Dewar, R.B.K., Dubinsky, E., Schonberg, E.: Programming with Sets: An Introduction to SETL. Texts and Monographs in Computer Science. Springer (1986)
24. Smith, D.R.: KIDS: A semi-automatic program development system. IEEE Transactions on Software Engineering, 16(9):1024–1043, (September 1990)
25. Smith, D.R., Kotik, G.B., Westfold, S.J.: Research on knowledge-based software environments at Kestrel Institute. IEEE Transactions on Software Engineering, 11(11):1278–1295, (November 1985)
26. Wirth, N.: Program Development by Stepwise Refinement. Communications of the ACM, 14:221–227 (1971)

Precise Documentation: The Key to Better Software

David Lorge Parnas

Middle Road Software, Inc.

Abstract. The prime cause of the sorry "state of the art" in software development is our failure to produce good design documentation. Poor documentation is the cause of many errors and reduces efficiency in every phase of a software product's development and use. Most software developers believe that "documentation" refers to a collection of wordy, unstructured, introductory descriptions, thousands of pages that nobody wanted to write and nobody trusts. In contrast, Engineers in more traditional disciplines think of precise blueprints, circuit diagrams, and mathematical specifications of component properties. Software developers do not know how to produce precise documents for software. Software developments also think that documentation is something written after the software has been developed. In other fields of Engineering much of the documentation is written before and during the development. It represents forethought not afterthought. Among the benefits of better documentation would be: easier reuse of old designs, better communication about requirements, more useful design reviews, easier integration of separately written modules, more effective code inspection, more effective testing, and more efficient corrections and improvements. This paper explains how to produce and use precise software documentation and illustrate the methods with several examples.

1 Documentation: a perpetually unpopular topic

This paper presents the results of many years of research on a very unpopular subject. Software documentation is disliked by almost everyone.

- Program developers don't want to prepare documentation; their lack of interest is evident in the documents that they deliver.
- User documentation is often left to technical writers who do not necessarily know all the details. The documents are often initially incorrect, inconsistent and incomplete and must be revised when users complain.
- The intended readers find the documentation to be poorly organized, poorly prepared and unreliable; they do not want to use it. Most consider "try it and see" or "look at the code" preferable to relying on documentation.
- User documentation is rapidly being replaced by "help" systems because it is hard to find the details that are sought in conventional documentation. Unfortunately, the "help" system answers a set of frequently occurring questions and is usually incomplete and redundant. Those who have an unusual question don't get much help.

S. Nanz (ed.), *The Future of Software Engineering*,
DOI 10.1007/978-3-642-15187-3_8,

These factors feed each other in a vicious cycle.

- Reduced quality leads to reduced usage.
- Reduced usage leads to reductions in both resources and motivation.
- Reduced resources and motivation degrade quality further.

Most Computer Science researchers do not see software documentation as a topic within computer science. They can see no mathematics, no algorithms, and no models of computation; consequently, they show no interest in it. Even those whose research approach is derived from mathematics, do not view what they produce as documentation. As a consequence of failing to think about how their work could be used, they often produce notations that are not useful to practitioners and, consequently, not used.

This paper argues that the preparation of well-organized, precise documentation for use by program developers and maintainers is essential if we want to develop a disciplined software profession producing trustworthy software. It shows how basic mathematics and computer science can be used to produce documentation that is precise, accurate, and (most important) useful. The first section of the paper discusses why documentation is important. The main sections show how we can produce more useful documents. The final section discusses future research and the transfer of this research to practice.

2 Types of documents

Any discussion of documentation must recognize that there are many types of documents, each useful in different situations. This section introduces a number of distinctions that are important for understanding the basic issues.

2.1 Programming vs. software design

Newcomers to the computer field often believe that "software design" is just another name for programming. In fact, programming is only a small part of software development [30]. We use the term "software" to refer to a program or set of programs written by one group of people for repeated use by another group of people. This is very different from producing a program for one's own use or for a single use.

- When producing a program for one's own use, you can expect the user to understand the program and know how to use it. For example, that user (yourself) will be able to understand the source of an error message or the cause of some other failure. There is no need to offer a manual that explains what parameters mean, the format of the input, or compatibility issues. All of these are required when preparing a program that will be used by strangers. If the user is the creator, questions that arise because he or she has forgotten the details can be answered by studying the code. If a non-developer user forgets something that is not in the documents, they may have no alternative but to find someone who knows the code.

– When producing a program for a single use, there is no need to design a program that will be easy to change or easily maintained in several versions. In contrast, programs that are successful software products will be used for many years; several versions may be in use simultaneously. The cost of updating and improving the program often greatly exceeds the original development costs. Consequently, it is very important to design the program so that it is as easy to change as possible and so that members of the software product line have as much in common as possible. These commonalities must be documented so that they remain common as individual family members are revised.

It is the differences between software development and programming (multi-person involvement, multi-version use) that make documentation important for software development.

2.2 What is a document?

A *description* of a system is accurate information about the system in written or spoken form. Usually, we are concerned with written descriptions, descriptions that will exist for a substantial period of time.

A *document* is a written description that has an official status or authority and may be used as evidence. In development, a document is usually considered binding, i.e. it restricts what may be created. If deviation is needed, revisions of the document must be approved by the responsible authority.

A document is expected to be correct or accurate, i.e. it is expected that the information that one may glean from the document is actually true of the system being described. When this is not the case, there is an error: either the system is defective, the document is incorrect, or both are wrong. In any case, the purported document is not a description of the system. Such faulty descriptions are often still referred to as "documents" if they have an official status.

Documents are expected to be precise and unambiguous. In other words, there should be no doubt about what they mean. However, even if they are imprecise or unclear, they may still be considered to be documents.

It should be noted that individual documents are not expected to be complete in the sense of providing all of the information that might be given about the system. Some issues may not have been resolved at the time that a document was written or may be the subject of other documents[1]. However, the content expected in a class of documents should be specified. For example, food packages are required to state certain specific properties of their contents. When there is a content-specification, documents are expected to be *complete relative to that specification*.

In connection with software, the word "document" is frequently used very sloppily. Information accompanying software often comes with disclaimers that warn that they are not necessarily accurate or precise descriptions of the software. Where one might expect statements of responsibility, we find disclaimers.

[1] It is generally bad practice to have the same information in more than one document as this allows the documents to become inconsistent if some are changed and others are not.

2.3 Are computer programs self-documenting?

One of the reasons that many do not take the problem of software documentation seriously is that the code itself looks like a document. We view it on paper or on a screen using exactly the same skills and tools that we would use for a reading a document. In 2006, Brad Smith, Microsoft Senior Vice President and General Counsel, said. "The Windows source code is the ultimate documentation of Windows Server technologies" [33].

No such confusion between the product and the documentation occurs when we develop physical artifacts; there is a clear difference between a circuit diagram and the circuit or between a bridge and its documentation. We need separate documents for physical products such as buildings and electronic circuits. Nobody wants to crawl around a building to find the locations of the ducts and pipes. We expect to find this information in documents.[2]

It is quite common to hear code described as self documenting; this may be true "in theory" but, in practice, most programs are so complex that regarding them as their own documentation is either a naive illusion or disingenuous. We need documents that extract the essential information, abstracting from the huge amounts of information in the code that we do not need. Generally, what is needed is not a single document but a set of documents, each giving a different view of the product. In other words, we have to ignore the fact that we could print out the code as a set of characters on some medium and provide the type of documentation that is routinely provided for physical products.

2.4 Internal documentation vs. separate documents

With physical products it is clear that we do not want the documentation distributed throughout the product. We do not want to climb a bridge to find out what size the nuts and bolts are. Moreover, we do not want to have to understand the bridge structure to decide whether or not we can drive our (heavily loaded) vehicle across. We expect documentation to be separate from the product and to provide the most needed abstractions (e.g. capacity).

Some researchers propose that specifications, in the form of assertions [12] or program functions [11], be placed throughout the code. This may be useful to the developers but it does not meet the needs of other readers. Someone who wants to use a program should not have to look in the code it to find out how to use it or what its capabilities and limits are. This is especially true of testers who asked to prepare "black box" test suites in advance of completion so that they can start their testing as soon as the code is deemed ready to test.

[2] Our expectations are not always met; sometimes someone has not done their job.

2.5 Models vs. documents

Recognition of a need for something other than the code has sparked an interest in models and approaches described as model-driven engineering. It is important to understand the distinction between "model "and "document".

A *model* of a product is a simplified depiction of that product; a model may be physical (often reduced in size and detail) or abstract. A model will have some important properties of the original. However, not all properties of the model are properties of the actual system. For example, a model airplane may be made of wood or plastic; this is not true of the airplane being modelled. A model of a queue may not depict all of the finite limitations of a real queue. In fact, it is common to deal with the fact that we can build software stacks with a variety of capacity limits by using a model that has unlimited capacity. No real stack has this property.

A *mathematical model* of a system is a mathematical description of a model; that mighty not actually exist. A mathematical model may be used to calculate or predict properties of the system that it models.

Models can be very useful to developers but, because they are not necessarily accurate descriptions, they do not serve as documents. One can derive information from some models that will not be true of the real system. For example, using the model of an unlimited stack, one may prove that the sequence PUSH;POP on a stack will leave it unchanged. This theorem is not true for real stacks because they might be full before the PUSH and the subsequent POP will remove something that was there before.

It follows that models must be used with great care; the user must always be aware that any information obtained by analyzing the model might not apply to the actual product. Only if the model has no false implications, can it be treated as a document.

However, every precise and accurate document can serve as a safe mathematical model because a document is usually simpler than the real product and has the property that everything you can derive from the document is true of the actual object. Precise documents can be analyzed and simulated just as other models can.

2.6 Design documents vs. introductory documentation

When a document is prepared, it may be intended for use either as a tutorial narrative or as a reference work.

- Tutorial narratives are usually designed to be read from start to end.
- Reference works are designed to help a reader to retrieve specific facts from the document.
- Narratives are usually intended for use by people with little previous knowledge about the subject.
- Reference documents are generally designed for people who already know a lot about the subject but need to fill specific gaps in their knowledge.

The difference may be illustrated by contrasting introductory language textbooks with dictionaries for the same language. The textbooks begin with the easier and more

fundamental characteristics of the language. Dictionaries arrange words in a specified (but arbitrary) order that allows a user to quickly find the meaning of a specific word.

Generally, narratives make poor reference works and reference works are a poor way to get an introduction to a subject.

In the software field we need both kinds of documents. New users and developers will need to receive a basic understanding of the product. Experienced users and developers will need reference documents.

This paper is about reference documents. The preparation of introductory narratives is a very different task.

2.7 Specifications vs. other descriptions

When preparing engineering documents, it is important to be conscious of the role that the document will play in the development process. There are two basic roles, description and specification.

- *Descriptions* provide properties of a product that exists (or once existed).
- S*pecifications*[3] are descriptions that state only required properties of a product that might not yet exist.
- A specification that states all required properties is called a *full specification*.
- General descriptions may include properties that are incidental and not required.
- Specifications should only state properties that are required. If a product does not satisfy a specification it is not acceptable for the use intended by the specification writer.

The difference between a specification and other descriptions is one of intent, not form. Every specification that a product satisfies is a description of that product. The only way to tell if a description is intended to be interpreted as a specification is what is said about the document, not its contents or notation.

The distinction made here is important whenever one product is used as a component of another one. The builder of the using product should be able to assume that any replacements or improvements of its components will still have the properties stated in a specification. That is not true if the document is a description but not a specification. Unfortunately, many are not careful about this distinction.

A specification imposes obligations on the implementers, users, and anyone who requests a product that meets the specification.

When presented with a specification, *implementers* may either

- accept the task of implementing that specification, or
- reject the job completely, or
- report problems with the specification and propose a revision.

They may not accept the task and then build something that does not satisfy the specification.

[3] In this paper, we use the word “specification” as it is traditionally used in Engineering, which is different from the way that it has come to be used in Computer Science. In Computer Science, the word is used, often without definition, to denote any model with some (often unspecified) relation to a product.

- *Users* must be able to count on the properties stated in a specification; they must not base their work on any properties stated in another description unless they are also stated in the specification.
- *Purchasers* are obligated to accept (and pay for) any product that meets the (full) specification included in a purchase agreement or bid.

Other descriptions may be useful for understanding particular implementations, for example additional descriptive material may document the behaviour of an implementation in situations that were treated as "don't care" cases in the specification.

2.8 Extracted documents

If the designers did not produce the desired documentation, or if the document was not maintained when the product was revised, it is possible to produce description documents by examining the product. For example, the plumbing in a building may be documented by tracing the pipes. If the pipes are not accessible, a die can be introduced in the water and the flow determined by observing the colour of the water at the taps.

There are many things that documentation extracted from the product cannot tell you. Documents produced by examining a product will be descriptions but not usually specifications. Extracted documents cannot tell you what was intended or what is required. Describing the actual structure is valuable in many circumstances but not a substitute for a specification.

Extracted documents usually contain low-level facts rather than the abstractions that would be of more value. The derivation of useful abstractions from construction details is a difficult problem because it requires distinguishing between essential and incidental aspects of the code.

Extracted documentation is obviously of no value during development, e.g. for design reviews or to improve communication between developers. It is also not a valid guide for testers because one would be testing against a description, not a specification. Testing on the basis of derived documents is circular; if you do this, you are assuming that the code is correct and testing to see that it does what it does.

Because software is always in machine-readable form, there are many tools that process code, extracting comments and declarations, and assemble these abstracts into files that are called documents. These extracted documents mix irrelevant internal details with information that is important for users; these documents will not usually be good documents for users, and reviewers, or maintainers. They include mostly the nearly random selection of information that a programmer chose to put in the comments.

These tools are apparently intended for people who are required to produce documentation but do not want to do the work required. Their major advantage is that the documentation produced can be revised quickly and automatically whenever the program is changed.

3 Roles played by documents in development

There are a number of ways that precise documentation can be used during the development and maintenance of a software product. The following sections describe the various ways that the software will be used.

3.1 Design through documentation

Design is a process of decision making; each decision eliminates one or more alternative designs. Writing a precise design document forces designers to make decisions and can help them to make better ones. If the documentation notation is precise, it will help the designers to think about the important details rather than evade making conscious decisions. A design decision will only be able to guide and constrain subsequent decisions if it is clearly and precisely documented. Documentation is the medium that designers use to record and communicate their decisions [6].

3.2 Documentation based design reviews

Every design decision should be reviewed by other developers as well as specialists in specific aspects of the application and equipment. Highly effective reviews can be based on the design documentation. If the documentation conforms to a standard structure, it can guide the review and make it less likely that important aspects of the decision will be overlooked [20].

3.3 Documentation based code inspections

A document specifying what that code should do is a prerequisite for a useful inspection of the actual code. If the code is complex, a "divide and conquer" approach, one that allows small units to be inspected independently of their context will be needed. A specification will be needed for each of those units. The specification for a unit, M, is used at least twice, once when inspecting M itself and again when inspecting units that use M. This is discussed in more depth in [21,22,23,26,29].

3.4 Documentation based revisions

Software that proves useful will, almost inevitably, be changed. Those who make the changes will not necessarily be the people who created the code and even if they are the same people they will have forgotten some details. They will need reliable information. Well-organized documentation can reduce costs and delays in the maintenance phase.

3.5 Documentation in contracts

An essential part of every well-written development contract is a specification that characterizes the set of acceptable deliverables. Specifications should not be confused with the contract. The contract also includes schedules, cost formulae, penalty clauses, statements about jurisdictions for dispute settlement, warranty terms, etc.

3.6 Documentation and attributing blame

Even in the best development projects, things can go wrong. Unless there is clear documentation of what each component must do, a 'finger-pointing' exercise, in which every group of developers blames another, will begin. With precise documentation, the obligations of each developer will be clear and blame (as well as costs) can be assigned to the responsible parties.

3.7 Documentation and compatibility

The chimera of interchangeable and reusable components is sought by most modern software development methods. Compatibility with a variety of components does not happen by accident; it requires a clear statement of the properties that the interchangeable components must share. This interface documentation must be precise; if an interface specification is ambiguous or unclear, software components are very unlikely to be interchangeable.

4 Costs and benefits of software documentation

Everyone is conscious of the cost of producing software documents because production costs are relatively easy to measure. It is much harder to measure the cost of *not* producing the documentation. This cost is buried in the cost of making changes, the cost of finding bugs, the cost of security flaws, the costs of production delays, etc. As a consequence it is hard to measure the benefits of good documentation. This section reviews some of those hidden costs and the associated benefits.

Losing time by adding people
In 1975, in a brilliant book of essays [2], Frederick P. Brooks, Jr. warned us that adding new staff to a late project could make it later. Newcomers need information that is often easily obtained by asking the original staff. During the of period when the newcomers are learning enough about the system to become useful, those who do understand the system spend much of their time explaining things to the newly added staff and, consequently, are less productive than before.

This problem can be ameliorated by giving the newcomers design documentation that is well-structured, precise, and complete. The newcomers can use this

documentation rather than interrupt the experienced staff. The documentation helps them to become useful more quickly.

Wasting time searching for answers
Anyone who has tried to change or debug a long program knows how long it can take to find key details in unstructured documentation or undocumented code. After the software has been deployed and is in the maintenance (corrections and updates) phase, those who must make the changes often spend more than half[4] of their time trying to figure out how the code was intended to work. Understanding the original design concept is a prerequisite to making safe changes. It is also essential for maintaining design integrity and efficiently identifying the parts of the software that must be changed.

Documentation that is structured so that information is easy to retrieve can save many frustrating and tiring hours.

Wasting time because of incorrect and inconsistent information
Development teams in a rush often correct code without making the corresponding changes in documentation. In other cases, with unstructured documentation, the documentation is updated in some places but not in others. This means that the documents become inconsistent and partially incorrect. Inconsistency and errors in documentation can lead programmers to waste time when making corrections and additions.

The cost of undetected errors
When documentation is incomplete and unstructured, the job of those assigned to review a design or to inspect the code is much harder. This often results in errors in releases and errors that are hard to diagnose. Errors in a released and distributed version of a product have many costs for both user and developer. Among other things, the reputation of the developer suffers.

Time wasted in inefficient and ineffective design reviews
During design reviews, much time is wasted while reviewers listen to introductory presentations before than can look for design errors. In fact, without meaningful design documents, design reviews often degenerate to a mixture of bragging session and tutorial. Such reviews are not effective at finding design errors. An alternative approach is active design reviews based on good documentation as described in [20].

Malicious exploitation of undocumented properties by others
Many of the security flaws that plague our popular software today are the result of a hacker discovering an undocumented property of a product and finding a way to use it. Often a security flaw escapes the attention of reviewers because they rely on the documentation and do not have the time to study the actual code. Many would be discovered and eliminated if there was better documentation.

[4]This is an estimate and is based on anecdotes.

5 Considering readers and writers

The first step towards producing better documentation is to think about who will read it, who will write it, what information the readers will need and what information the writers will have when they are writing.

Engineering documentation comprises many separate documents rather than a single, "all-in-one" document because there is a great variety of readers and writers. The readers do not all have the same needs or the same expertise. Each writer has different information to communicate. A developer's documentation standards must be designed with the characteristics of the expected readers and writers in mind.

Figure 1 lists a number of important software documents and the intended authors and audience for each.

Document	Writers	Readers/Users
Software Requirements Document	User reps, UI experts, application experts, controlled hardware experts	Authors of module guide and module interface specifications, (Software "Architects")
Module Guide	Software "Architects"	All Developers
Module Interface Specifications	Software "Architects"	Developers who implement or use the module
Program Uses Structure	Software "Architects"	Component Designers, Programmers
Module Implementation Design Document	Component Designers	Programmers implementing component
Display Method Program Documentation	Programmers implementing component	inspectors, maintainers potential reusers

Fig. 1. Software Document Readers and Writers

It is important to note that no two documents have the same readers or creators. For example, the people who have the information needed to determine the system requirements are not usually software developers. In an ideal world, representatives of the ultimate users would be the authors of requirements documentation. In today's world, developers might have to "ghost write" the document for those experts after eliciting information from them. This is far from ideal but it is a fact.

In contrast, determining the module structure of a product *is* a matter for software developers and, if the project has created a good[5] software requirements[6] document, the user representatives should not need to provide input or even review the module structure. Given a precise and complete module interface specification, the developers who implement the module need not ever consult the requirements documents, the

[5] The next section describes what is meant by "good".

[6] Sadly, most current requirements documents are not good enough to allow this. To do this, they would have to restrict the visible behaviour so precisely that any implementation that satisfied the requirements document would satisfy the users.

people who wrote those documents, or the user representatives. Unfortunately, the state of the art has not yet reached that nirvana.

The first step in producing good software documentation is to gain a clear picture of the authors and readers, their expertise and information needs.

The job titles that appear in the summary table (Figure 1) are buzzwords without precise definition. For example, "Software Architect" is almost meaningless. The ability to design interfaces is distinct from the ability to find a good internal structure for the software but a Software Architect may have either of those skills or both. Each development organization will have to produce its own version of Figure 1 based on its own job titles and associated skill descriptions.

With a set of readers identified, one must then ask how they will use a document, i.e. what questions they will use the document to answer. The document must be organized so that those questions will be easy to answer. Often several readers will have the same information needs but will ask different questions. For example, information about a component's interface will be needed by those who develop the component, those who test the component, those who use the component, and those assigned to maintain the component but they may be asking different questions. For example, a user may ask, "How do I get this information on the screen?", a tester or maintainer may ask, "When does this information appear on the screen?", a developer will be asking "What should the program display in these circumstances?". It is tempting to give each reader a separate document but this would be a mistake as they may get different information. The document should be structured so that any of those groups can answer their questions easily.

6 What makes design documentation good?

Above we have spoken of "good" design documentation. It is important to be precise about what properties we expect. The four most important are accuracy, lack of ambiguity, completeness, and ease of access.

6.1 Accuracy

The information in the document must be true of the product. Misinformation can be more damaging than missing information. Accuracy is most easily achieved if one can test that the product behaves as described. A precise document can be tested to see whether or not the product has the properties described in the document.

6.2 Unambiguous

The information must be precise, i.e. have only one possible interpretation. When reader and writer, (or two readers) can interpret a document differently, compatibility problems are very likely. Experience shows that it is very hard to write unambiguous documents in any natural language.

6.3 Completeness

As explained earlier, one should not expect any one document to describe everything about a product; each presents one view of the product. However, each view should be complete, i.e. all the information needed for that view should be given. For example, a black box description should describe the possible behaviour for all possible input sequences and a data structure description should include all data used by the product or component. A document format is good if it makes it easy to demonstrate completeness of a description.

6.4 Ease of access

With the complexity of modern software systems completeness is not enough. It must be possible to find that information quickly. This requires that clear organizational rules apply to the document. The rules should dictate where information is put in a document and can be used to retrieve information. There should be one and only one place for every piece of information.

7 Documents and mathematics

It is rare to hear anyone speak of software documentation and mathematics together. It is assumed that documents are strings in some natural language. In fact, documents are inherently mathematical and using mathematics in documentation is the only way to produce good documents.

7.1 Documents are predicates

We can examine a product, P, and a document to see if everything we can derive from the document is true of the product. There are 3 possibilities:

- The document is an accurate description of P.
- The document is not an accurate description of P.
- The document is meaningless or undefined when applied to P.[7]

In this way, a document partitions the set of products into three sets, those products for which it is *true*, those for which it is *false*, and those for which it is undefined. We can regard the document as a predicate; the domain of that predicate is the union of the first two sets. The third set comprises elements that are outside of the domain of P. If we wish to use the logic described in [25], we extend the predicate P to a total predicate P' by evaluating P' to *false* where P is not defined.

By treating documents as predicates, we can write "document expressions" to characterize classes of products.

[7] For example, a document that specified the colour of a product is meaningless for software products.

7.2 Mathematical definitions of document contents

Organizations that are serious about design documentation organize reviews for the documents that are produced. Many of the disagreements that arise during those reviews are about what information should be in the document, not about the design decisions that are made in the document. For example, some reviewers might argue that a data structure interface does not belong in a requirements document since it is a design decision. Others might suggest that this data will be communicated to other systems and the information about its representation is an essential part of the system requirements. Such disagreements are counterproductive as they distract from the real issues. Neither side would be considering the merit of the data structure itself. To minimize the time lost on such discussions, it is necessary to provide precise definitions of the content (not the format) of the documents.

A method of doing this is described in [24]. There it was shown that most design documents can be understood as representations of specific relations, for example the relation between input and output values. We will do this for a few key documents below.

7.3 Using mathematics in documents

To achieve the accuracy, lack of ambiguity, completeness, and ease of access that we expect of engineering design documents, natural language text should be avoided; the use of mathematics instead is essential. Once the contents of a document is defined abstractly as a set of relations, it is necessary to agree on how to represent those relations. A document such as the requirements document [4], described in [5], can consist of a set of mathematical expressions. These expressions represent the characteristic predicate of the relations being represented by the document. Representations that make these expressions easier to read, review, and write were used in [4] and are discussed and illustrated in section 9.

8 The most important software design documents

Each project will have its own documentation requirements but there is a small set of documents that is always needed. They are:

- The systems requirements document
- The module structure document
- Module interface documents
- Module internal design documents
- Program function documents

8.1 Requirements documentation

The most fundamental obligation of a Professional Engineer is to make sure that their products are fit for use. This clearly implies that the Engineer must know what the requirements are. These requirements are not limited to the conscious wishes of the customer; they include other properties of which the customer might not be aware and requirements implied by the obligation of Engineers to protect the safety, well-being and property of the general public.

To fulfill their obligations, Engineers should insist on having a document that clearly states all requirements and is approved by the relevant parties. An Engineer must insist that this document is complete, consistent, and unambiguous.

No user visible decisions should be left to the programmers because programmers are experts in programming but often do not have expertise in an application or knowledge of the needs of users. Decisions about what services the user should receive should be made, documented, and reviewed by experts. The resulting document should constrain the program designers.

The two-variable model for requirements documentation

The two-variable model is a simple model that has been used in many areas of science and engineering. It is based on Figure 2.

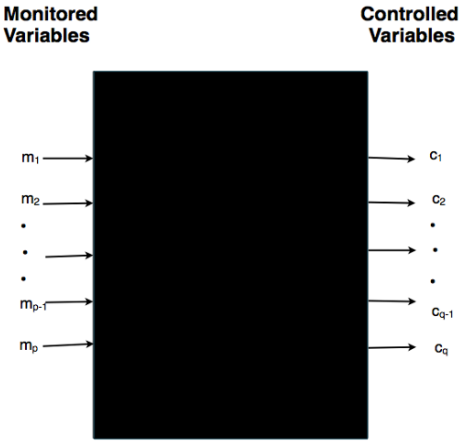

Fig. 2. Black Box View used for two-variable model.

A product can be viewed as a black box with p inputs and q outputs.

- We are given no information about the internals.
- The outputs, often referred to as *controlled variables*, $c_1,\ldots, c_q$, are variables whose value is determined by the system.
- The inputs, often known as *monitored variables*, $m_1,\ldots, m_p$, are variables whose value is determined externally.

- MC denotes $(m_1, \ldots m_p, c_1, \ldots c_q)$
- M(MC) and C(MC) denote $m_1, \ldots m_p$ and $c_1, \ldots, c_q$ respectively.
- If S is a set of MC-tuples M(S) is a set defined by $\{q \mid \exists T, T \ni S \wedge M(T) = q\}$ and C(S) is a set defined by $\{q \mid \exists T, T \ni S \wedge C(T) = q\}$.
- The values of a variable, v, over time can be described by a function of time denoted v^t. The expression $v^t(T)$ denotes the value of the v^t at time T.
- The values of any tuple of variables, V, over time can be described by a function of time denoted V^t. $V^t(T)$ denotes the value of the V^t at time T.

Viewing V^t as a set of ordered pairs of the form $(t, V^t(t))$:

- The subset of V^t with $t \leq T$ will be denoted V^t_T.

The product's possible behaviour can be described by a predicate, $SYSP(MC^t_T)$ where:

- MC^t_T is a history of the monitored and controlled values up to, and including, time T, and,
- $SYSP(MC^t_T)$ is *true* if and only if MC^t_T describes possible behaviour of the system.

This formulation allows the description of products such that output values can depend immediately on the input values (i.e., without delay). It also allows describing the values of the outputs by stating conditions that they must satisfy rather than by specifying the output as a function of the input[8].

It is sometimes convenient to use a relation, SYS derived from SYSP. The domain of SYS contains values of M^t_T and the range contains values of C^t_T.

- $SYS = \{(m, c) \mid \exists mc, SYSP(mc) \wedge C(mc) = c \wedge M(mc) = m\}$

In deterministic systems, the output values are a function of the input; consequently, the values of outputs in the history are redundant. In deterministic cases, we can treat SYS as a function with a domain comprising values of M^t_T and a range comprising values of C^t_T and to write $SYS(M^t_T)$ in expressions. $SYS(M^t_T)(T)$ evaluates to the value of the outputs at time T.

In the non-deterministic case, there are two complicating factors:

- The history need not determine a unique value for the output; SYS would not necessarily be a function.
- The output at time t may be constrained by previous output values, not just the input values.[9]

For these reasons, in the general (not necessarily deterministic) case the output values have to be included in history descriptions.

If the product is deterministic, SYS will be a function. In the non-deterministic case, the relation must either be used directly (e.g. as a predicate) or one can construct

[8] It is possible to describe conditions that cannot be satisfied. Care must be taken to avoid such nonsense.

[9] A simple example to illustrate this problem is the classic probability problem of drawing uniquely numbered balls from an opaque urn without replacing a ball after it is removed from the urn. The value that has been drawn cannot be drawn again, but except for that constraint, the output value is random.

a function whose range includes sets of possible output values rather than individual values.

For a 2-variable system requirements document we need two predicates NATP and REQP.

Relation NAT

The environment, i.e. the laws of nature and other installed systems, places constraints on the values of environmental quantities. They can be described by a predicate, NATP(MC^t_T), where:

- MC^t_T is a history of the input and output variable values up to, and including, time, T and,
- NATP(MC^t_T) if and only if MC^t_T describes possible behaviour of the product's environment if the product is not active in that environment.

It is sometimes convenient to represent NATP as a relation, NAT, with domain comprising values of MC^t_T and range comprising values of C^t_T. This is defined by:

- $NAT = \{(m, c) \mid \exists mc, NATP(mc) \wedge C(mc) = c \wedge M(mc) = m\}$

Relation REQ

No product can override NAT; it can only impose stricter constraints on the values of the output variables. The restrictions can be documented by a predicate, REQP(MC^t_T), where:

- MC^t_T is a history of the monitored and controlled variable values up to, and including, time T and,
- REQP(MC^t_T) is *true* if and only if MC^t_T describes permissible behaviour of the system.

It is sometimes convenient to represent REQP as a relation, REQ, with domain comprising values of MC^t_T and range comprising values of $C^t(T)$. This is defined by:

- $REQ = \{(m, c) \mid \exists mc, REQP(mc) \wedge C(mc) = c \wedge M(mc) = m\}$

If deterministic behaviour is required, we can write REQ (M^t_T), a function with values in the range, $C^t(T)$.

Experience and examples

Numerous requirements documents for critical systems have been written on the basis of this model. The earliest [4] are described in [5].

8.2 Software component interface documents

The two-variable model described above can be applied to software components, especially those that have been designed in accordance with the information hiding principle [13,14,18]. Because software components change state only at discrete points in time, a special case of the two-variable model, known as the Trace Function Method (TFM) can be used. TFM has been found useful for several industrial

products as described in [1,34,31,32]. These documents are easily used as referenced documents, can be checked for completeness and consistency, and can be input to simulators for evaluation of the design and testing of an implementation. The process of preparing these documents proved their usability; although they were developed by researchers, they were reviewed by practitioners who reported many detailed factual errors in the early drafts. If people cannot or will not read a document, they will not find faults in it.

8.3 Program function documents

Those who want to use a program do not usually want to know how it does its job; they want to know what job it does or is supposed to do. It has been observed by many mathematicians that a terminating deterministic program can be described by a function mapping from a starting state to a stopping state. Harlan Mills built an original approach to structured programming on this observation [11]. States were represented in terms of the values of program variables in those states. This was used extensively within IBM's Federal Systems Division. The main problem encountered was that developers found it difficult to represent the functions and often resorted to informal statements.

Other mathematicians have observed that one can describe non-deterministic programs with a relation from starting state to stopping states by adding a special element of the stopping state set to represent non-determination. This approach was perfectly general in theory but in practice, it proved difficult to represent the non-termination pseudo state in terms of program variable values.

A mathematically equivalent approach using two elements, a set of states in which termination was guaranteed and a relation was proposed by Parnas [17]. In this approach, both the set and the relation can be described in terms of program variable values making it easier to use the method in practice.

The remaining problem was the complexity of the expressions. This problem can be ameliorated by using tabular expressions (see below). The use of program function tables as program documentation has been illustrated in [27,26]. A further discussion of tabular expressions will be found below.

8.4 Module internal design documents

Many authors have noted that the design of a software component that has a hidden (internal) data structure can be defined by describing

- the hidden internal data structure,
- the program functions of each externally accessible program, i.e. their effect on the hidden data structure,
- an abstraction relation mapping between internal states and the externally distinguishable states of the objects created by the module.

The data structure can be described by programming language declarations. The functions are usually best represented using tabular expressions.

8.5 Documenting nondeterminism

One needs to be very careful when documenting programs that have non-deterministic behaviour.

We use the term “non-determinism” to describe situations where the output of a component is not-determined by the information available to us. For example, if a service can be provided by several different servers, each with satisfactory function but differences in performance, and we do not know which server we will get, we have non-determinism. If we had the information needed to predict which of the servers would be assigned to us, the behaviour would be deterministic.

Non-determinism is not the same as random behaviour. It is a matter of debate whether or not randomness exists but non-determinism exists without a doubt. The behaviour would be completely predictable by an observer that had the necessary information.

Non-determinism should not be confused with situations where there are alternatives that are equally acceptable but an acceptable alternative must be deterministic. For example, if purchasing a square-root finding program we might be happy with either a program that always gives the non-negative root or a program that always gives the negative root but we would not want one that alternated between the two.

Non-determinism can be documented by a relation. Choice as described in the previous paragraph requires a set of relations, one for each acceptable alternative.

When either non-determinism or choice is present it is often possible to write a using program that will get the desired result in any of the permitted cases. Documentation of the possible behaviour is essential for this.

Non-determinism is not an excuse for not providing precise documentation. The possible behaviours are always restricted and the set of possible characteristics must be characterized. In some cases there may be statistical information available and this too should be documented.

The same techniques that can be used for dealing with non-determinism can be used when developing software before some decisions about other programs or the environment have been made.

8.6 Additional documents

In addition to the system requirements document, which treats hardware and software as an integrated single unit, it is sometimes useful to write a software requirements document which is based on specific details about the input/output connections [24].

An informal document known as the module guide is also useful for larger systems. Hierarchically organized it classifies the modules by their “secrets” (the information hidden by each [18].

A uses relation document, which indicates which programs are used by each program is generally useful [16]. The information is a binary relation and may be represented in either tabular or graphical form.

In systems with concurrency, process structure documents are useful. The "gives work to" document is useful for deadlock prevention [15]. Interprocess/component communication should also be documented as discussed in [10].

9 Tabular expressions for documentation

Mathematical expressions that describe computer systems can become very complex, hard to write and hard to read. The power of software is its ability to implement functions that have many special cases. When these are described by expressions in conventional format, the depth of nesting of subexpressions gets to high. As first demonstrated in [5,4], the use of a tabular format for mathematical expressions can turn an unreadable symbol string into an easy to access complete and unambiguous document.

Figure 3 shows an expression that describes[10] the required behaviour of the keyboard checking program that was developed by Dell in Limerick, Ireland and described in [1,31].

Keyboard Checker: Conventional Expression

(N(T)=2∧keyOK∧(¬(T=_)∧N(p(T))=1))∨(N(T)=1∧(T=_∨(¬(T=_)∧N(p(T))=1))∧
(¬keyOK∧¬prevkeyOK∧¬prevkeyesc))∨((¬(T=_)∧N(p(T))=1)∧
((¬keyOK∧keyesc∧¬prevkeyesc)∨(¬keyOK∧keyesc∧prevkeyesc∧
prevexpkeyesc))∨((N(T)=N(p(T))+1)∧(¬(T=_)∧(1<N(p(T))<L))∧(keyOK))∨
((N(T)=N(p(T))-1))∧(¬keyOK∧¬keyesc∧(¬prevkeyOK∧prevkeyesc∧
preprevkeyOK)∨prevkeyOK)∧((¬(T=_)∧(1<N(p(T))<L))∨(¬(T=_)∧N(p(T))=L)))∨
((N(T)=N(p(T)))∧(¬(T=_)∧(1<N(p(T))≤L))∧((¬keyOK∧¬keyesc∧(¬prevkeyOK∧
prevkeyesc∧¬preprevkeyOK))∨(¬keyOK ∧¬prevkeyOK∧ ¬prevkeyesc)∨
(¬keyOK∧keyesc∧¬prevkeyesc)∨(¬keyOK∧keyesc∧prevkeyesc∧
prevexpkeyesc))∨((N(P(T)=Fail)∧(¬keyOK∧keyesc∧prevkeyesc∧
¬prevexpkeyesc)∧(1≤N(p(T))≤L))∨((N(P(T)=Pass)∧(¬(T= _)∧N(p(T))=L)∧(keyOK))

Fig. 3. Characteristic Predicate of a Keyboard Checker Program

Figure 4 is a tabular version of the same expression. It consists of 3 grids. For this type of expression, the top grid and left grid are called "headers". The lower right grid is the main grid. To use this type of tabular expression one uses each of the headers to

[10] Auxiliary predicates such as keyesc, keyOK, etc. are defined separately. Each is simply defined.

select a row and column. If properly constructed, only one of the column headers and one of the row headers will evaluate to *true* for any assignment of values to the variables. The selected row and column identify one cell in the main grid. Evaluating the expression in that cell will yield the value of the function for the assigned values of the variables. The tabular expression parses the conventional expression for the user. Instead of trying to evaluate or understand the complex expression in Figure 3, one looks at the simpler expressions in Figure 4. Generally, one will only need to evaluate a proper subset of those expressions.

N(T) =

keyOK			T=_	¬ (T= _) ∧ N(p(T))=1	¬ (T= _) ∧ 1<N(p(T))< L	¬ (T= _) ∧ N(p(T))= L
				2	N(p(T))+ 1	Pass
¬keyOK ∧	¬keyesc ∧	(¬prevkeyOK ∧ prevkeyesc ∧ preprevkeyOK) ∨ prevkeyOK			N(p(T)) - 1	N(p(T)) - 1
		¬prevkeyOK ∧ prevkeyesc ∧ ¬preprevkeyOK			N(p(T))	N(p(T))
		¬prevkeyOK ∧ ¬ prevkeyesc	1	1	N(p(T))	N(p(T))
	keyesc ∧	¬ prevkeyesc		1	N(p(T))	N(p(T))
		prevkeyesc ∧ ¬prevexpkeyesc		Fail	Fail	Fail
		prevkeyesc ∧ prevexpkeyesc		1	N(p(T))	N(p(T))

Fig. 4. Tabular expression equivalent to Figure 3.

There are many forms of tabular expressions. The grids need not be rectangular. A variety of types of tabular expressions are illustrated and defined in [9]. Although tabular expressions were successfully used without proper definition in a number of applications, precise semantics are needed for tools and full analysis. There have been four basic approaches to defining the meaning of these expressions. Janicki and his co-authors developed an approach based on information flow graphs that can be used to define a number of expressions [8]. Zucker based his definition on predicate logic. Khédri and his colleagues based their approach on relational algebra [3]. All three of these approaches were limited to certain forms of tables [35]. The most recent approach [9], which is less restricted, defines the meaning of these expressions by means of translation schema that will convert any expression of a given type to an equivalent conventional expression. This is the most general approach and provides a good basis for tools. The appropriate table form will depend on the characteristics of the function being described. Jin shows a broad variety of useful table types in [9] which provides a general approach to defining the meaning of any new type of table.

10 Summary and Outlook

This paper has argued that it is important to the future of software engineering to learn how to replace the poorly structured, vague and wordy design documentation that we use today with precise professional design documents. It has also shown that documents have a mathematical meaning and are composed of mathematical expressions; those expressions can be in a variety of tabular formats that have proven to be practical over a period of more than 30 years. While there is much room for improvement, and a variety of questions for researchers to answer, what we have today can, and should, be used. Unfortunately, it is rarely taught. Nonetheless successful applications in such areas as flight-software, nuclear power plants, manufacturing systems, and telephone systems show that we are at a point where it is possible and practical to produce better documentation this way.

From its inception, the software development has been plagued by a mentality that enjoys and glorifies the excitement of "cut and try" software development. Some programmers talk with a gleam in their eyes about the excitement of trying new heuristics, the exciting challenge of finding bugs. Many of them talk as if it is inherent in software that there will be many bugs and glitches and that we will never get it right. One said, "The great thing about this is that there is always more to do." One gets the impression that the view software development as a modern version of the old pinball machines. When you win, bells ring but when you lose there is just a dull thud and a chance to try again.

Over the same period most software developers have been quite disparaging about documentation and the suggestion that it should conform to any standards. I have been told, "You can criticize my code but I have the right to write my documentation the way I like", and, "If I had wanted to document, I would have been an English major". In fact, one of the current fads in the field (which has always been plagued by fashions described by buzzwords) advises people not to write any documentation but just focus on code.

It is time to recognize that we have become a serious industry that produces critical products. We can no longer view it as a fun game. We need to grow up. The first step towards maturity must be to take documentation seriously, and to use documentation as a design medium. When our documentation improves in quality, the software quality will improve too.

References

1. Baber, R., Parnas, D.L., Vilkomir, S., Harrison, P., O'Connor, T.: Disciplined Methods of Software Specifications: A Case Study. In: Proceedings of the International Conference on Information Technology Coding and Computing (ITCC'05). IEEE (2005)
2. Brooks, F.P. Jr.: The Mythical Man-Month: Essays on Software Engineering, 2nd Edition. Addison Wesley, Reading, MA (1995)
3. Desharnais, J., Khédri, R., Mili A.: Towards a Uniform relational semantics for tabular expressions. In: Proceedings of RelMiCS'98, pp. 53-57 (1998)
4. Heninger, K., Kallander, J., Parnas, D.L., Shore, J.,: Software Requirements for the A-7E Aircraft. Naval Research Laboratory Report 3876 (1978)

5. Heninger, K.L.: Specifying Software Requirements for Complex Systems: New Techniques and their Application. IEEE Transactions Software Engineering, vol. SE-6, pp. 2-13 (1980)
   - Reprinted as chapter 6 in [7]
6. Hester, S.D., Parnas, D.L., Utter, D.F.: Using Documentation as a Software Design Medium, Bell System Technical Journal 60(8), pp. 1941-1977 (1981)
7. Hoffman D.M., Weiss, D.M. (eds.): Software Fundamentals: Collected Papers by David L. Parnas. Addison-Wesley (2001)
8. Janicki, R.: Towards a formal semantics of Parnas tables. In: Proc. of 17th International Conference on Software Engineering (ICSE), pp. 231-240 (1995)
9. Jin, Y., Parnas, D.L.: Defining the meaning of tabular mathematical expressions. Science of Computer Programming 75(11), pp. 980-1000 (2010)
10. Liu, Z., Parnas, D.L., Trancón y Widemann, B.: Documenting and Verifying Systems Assembled from Components. Frontiers of Computer Science in China, Higher Education Press, co-published with Springer-Verlag, in press (2010)
11. Mills, Harlan D.: The New Math of Computer Programming. Communications of the ACM 18(1): 43-48 (1975)
12. Meyer, B.: Eiffel: The Language. Prentice-Hall (1991)
13. Parnas, D.L.: Information Distributions Aspects of Design Methodology. In: Proceedings of IFIP Congress '71, Booklet TA-3, pp. 26-30 (1971)
14. Parnas D.L.: On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM 15(12), pp. 1053-1058 (1972)
   - Translated into Japanese - BIT, vol. 14, no. 3, 1982, pp. 54-60.
   - Republished in Classics in Software Engineering, edited by Edward Nash Yourdon, Yourdon Press, 1979, pp. 141-150.
   - Republished in Great Papers in Computer Science, edited by Phillip Laplante, West Publishing Co, Minneapolis/St. Paul 1996, pp. 433-441.
   - Reprinted as Chapter 7 in [7]
   - Reprinted in Software Pioneers: Contributions to Software Engineering, Manfred Broy and Ernst Denert (Eds.), Springer Verlag, pp. 481 - 498 (2002)
15. Parnas, D.L.: On a 'Buzzword': Hierarchical Structure, IFIP Congress '74, North Holland Publishing Company, pp. 336-339 (1974)
   - Reprinted as Chapter 8 in [7]
   - Reprinted in Software Pioneers: Contributions to Software Engineering, Manfred Broy and Ernst Denert (Eds.), Springer Verlag, pp. 501 - 513 (2002)
16. Parnas, D.L.: Designing Software for Ease of Extension and Contraction. IEEE Transactions on Software Engineering, pp. 128-138 (1979)
   - Also in Proceedings of the Third International Conference on Software Engineering, pp. 264-277 (1978)
   - Reprinted as Chapter 14 in [7]
17. Parnas, D.L.: A Generalized Control Structure and its Formal Definition. Communications of the ACM 26(8), pp. 572-581 (1983)
18. Parnas, D.L., Clements, P.C., Weiss, D.M.: The Modular Structure of Complex Systems. IEEE Transactions on Software Engineering, vol. SE-11 no. 3, pp. 259-266 (1985)
19. Parnas, D.L., Wadge, W.: A Final Comment Regarding An Alternative Control Structure and its Formal Definition (Technical Correspondence). Communications of the ACM 27(5), pp. 499-522 (1984)
20. Parnas, D.L., Weiss, D.M.: Active Design Reviews: Principles and Practices. In: Proceedings of the 8th International Conference on Software Engineering (1985)
   - Also published in Journal of Systems and Software, December 1987.
   - Reprinted as Chapter 17 in [7].

21. Parnas, D.L., Asmis, G.J.K., Kendall, J.D.: Reviewable Development of Safety Critical Software. The Institution of Nuclear Engineers, International Conference on Control & Instrumentation in Nuclear Installations, paper no. 4:2 (1990)
22. Parnas, D.L., van Schouwen, A.J., Kwan, S.P.: Evaluation of Safety-Critical Software. Communications of the ACM 33(6), pp. 636-648 (1990)
    - Published in Advances in Real-Time Systems, John A. Stankovic and Krithi Ramamritham (editors), IEEE Computer Society Press, pp. 34-46 (1993)
    - Published in Ethics and Computing: Living Responsibly in a Computerized World, Kevin W. Bowyer (editor), IEEE Press, pp. 187-199 (2000)
23. Parnas D.L., Asmis, G.J.K., Madey, J.: Assessment of Safety-Critical Software in Nuclear Power Plants. Nuclear Safety 32(2), pp. 189-198 (1991)
24. Parnas, D.L., Madey, J.: Functional Documentation for Computer Systems Engineering. Science of Computer Programming 25(1), pp. 41-61 (1995)
25. Parnas D.L.: Predicate Logic for Software Engineering. IEEE Transactions on Software Engineering 19(9), pp. 856-862 (1993)
26. Parnas, D.L: Inspection of Safety Critical Software using Function Tables. In: Proceedings of IFIP World Congress 1994, Volume III, pp. 270 - 277 (1994)
    - Reprinted as Chapter 19 in [7]
27. Parnas, D.L., Madey, J., Iglewski, M.: Precise Documentation of Well-Structured Programs. IEEE Transactions on Software Engineering 20(12), pp. 948-976 (1994)
28. Parnas, D.L.: Teaching Programming as Engineering. In: The Z Formal Specification Notation, 9th International Conference of Z Users (ZUM'95), Lecture Notes in Computer Science vol. 967, Springer-Verlag, pp. 471-481 (1995)
    - Reprinted as Chapter 31 in [7]
29. Parnas, D.L.: Using Mathematical Models in the Inspection of Critical Software. In: Applications of Formal Methods. Prentice Hall International Series in Computer Science, pp. 17-31 (1995)
30. Parnas, D.L.: Structured programming: A minor part of software engineering. In: Special Issue to Honour Professor W.M. Turski's Contribution to Computing Science on the Occasion of his 65th Birthday, Information Processing Letters 88(1-2), pp. 53-58 (2003)
31. Parnas, D.L., Vilkomir, S.A: Precise Documentation of Critical Software. In: Proceedings of the Tenth IEEE Symposium on High Assurance Systems Engineering (HASE'07), pp. 237-244 (2007)
32. Parnas, D. L.: Document Based Rational Software Development. Knowledge-Based Systems 22(3), pp. 132-141 (2009)
33. Smith, B.: http://www.microsoft.com/presspass/press/2006/jan06/01-25EUSourceCodePR.mspx
34. Quinn, C., Vilkomir, S.A., Parnas, D.L. Kostic, S.: Specification of Software Component Requirements Using the Trace Function Method. In: Proceedings of the International Conference on Software Engineering Advances (ICSEA'06) (2006)
35. Zucker, J.I.: Transformations of normal and inverted function tables. Formal Aspects of Computing 8, pp. 679-705 (1996)

Empirically Driven Software Engineering Research

Dieter Rombach

Fraunhofer Institute for Experimental Software Engineering, Germany
University of Kaiserslautern, Germany
Dieter.Rombach@iese.fraunhofer.de

Abstract. Software engineering is a design discipline. As such, its engineering methods are based on cognitive instead of physical laws, and their effectiveness depends highly on context. Empirical methods can be used to observe the effects of software engineering methods in vivo and in vitro, to identify improvement potentials, and to validate new research results. This paper summarizes both the current body of knowledge and further challenges wrt. empirical methods in software engineering as well as empirically derived evidence regarding software typical engineering methods. Finally, future challenges wrt. education, research, and technology transfer will be outlined.

S. Nanz (ed.), *The Future of Software Engineering*,
DOI 10.1007/978-3-642-15187-3_9,

Component-based Construction of Heterogeneous Real-time Systems in BIP

Joseph Sifakis

VERIMAG
Joseph.Sifakis@imag.fr

Abstract. We present a framework for the component-based construction of real-time systems. The framework is based on the BIP (Behaviour, Interaction, Priority) semantic model, characterized by a layered representation of components. Compound components are obtained as the composition of atomic components specified by their behaviour and interface, by using connectors and dynamic priorities. Connectors describe structured interactions between atomic components, in terms of two basic protocols: rendezvous and broadcast. Dynamic priorities are used to select amongst possible interactions – in particular, to express scheduling policies.

The BIP framework has been implemented in a language and a toolset. The BIP language offers primitives and constructs for modelling and composing atomic components described as state machines, extended with data and functions in C. The BIP toolset includes an editor and a compiler for generating from BIP programs, C++ code executable on a dedicated platform. It also allows simulation and verification of BIP programs by using model checking techniques.

BIP supports a model-based design methodology involving three steps:

- The construction of a system model from a set of atomic components composed by progressively adding interactions and priorities.
- The application of incremental verification techniques. These techniques use the fact that the designed system model can be obtained by successive application of property-preserving transformations in a three-dimensional space: Behavior × Interaction × Priority.
- The generation of correct-by-construction distributed implementations from a BIP model. This is achieved by source-to-source transformations which preserve global state semantics.

We present the basic theoretical results about BIP including modelling interactions by using connectors, modelling priorities, incremental verification and expressiveness. We also present two examples illustrating the methodology as well as experimental results obtained by using the BIP toolset.

Further information is available at:
http://www-verimag.imag.fr/BIP,196.html

S. Nanz (ed.), *The Future of Software Engineering*,
DOI 10.1007/978-3-642-15187-3_10,

Computer Science: A Historical Perspective and a Current Assessment

Niklaus Wirth

wirth@inf.ethz.ch

Abstract. We begin with a brief review of the early years of Computer Science. This period was dominated by large, remote computers and the struggle to master the complex problems of programming. The remedy was found in programming languages providing suitable abstractions and programming models. Outstanding was the language Algol 60, designed by an international committee, and intended as a publication language for algorithms. The early period ends with the advent of the microcomputer in the mid 1970s, bringing computing into homes and schools. The outstanding computer was the Alto, the first personal computer with substantial computing power. It changed the world of computing.

In the second part of this presentation we consider the current state of the field and the shift of activities towards applications. Computing power and storage capacity seem to be available in unprecedented abundance. Yet there are applications that ask for even faster computers and larger stores. This calls for a new focus on multiprocessing, on systems with hundreds of processes proceeding concurrently. The invention of programmable hardware opens new possibilities for experimentation and exploring ways to design new architectures. Codesign of hardware and software becomes mandatory. The days of the general purpose von Neumann architecture seem to come to an end.

Finally, we will look at the present state of the art, the problems and the tools with which the engineer finds himself confronted today. Not only have computers become faster and storage bigger, but the tasks have become accordingly more demanding. The mass of tools has grown, programming is based on huge libraries, and the environments appear monstrously complicated and obscure. The most basic instruments, programming languages, have not improved. On the contrary, outdated languages dominate. They, and their tools, belong to the apparently irreplaceable legacy, and sometimes it seems that we have learnt little from the past. What can we do?

S. Nanz (ed.), *The Future of Software Engineering*,
DOI 10.1007/978-3-642-15187-3_11,

Internet Evolution and the Role of Software Engineering

Pamela Zave

AT&T Laboratories—Research, Florham Park, New Jersey, USA
pamela@research.att.com
WWW home page: http://www2.research.att.com/TILDEpamela

Abstract. The classic Internet architecture is a victim of its own success. Having succeeded so well at empowering users and encouraging innovation, it has been made obsolete by explosive growth in users, traffic, applications, and threats. For the past decade, the networking community has been focused on the many deficiencies of the current Internet and the possible paths toward a better future Internet. This paper explains why the Internet is likely to evolve toward multiple application-specific architectures running on multiple virtual networks, rather than having a single architecture. In this context, there is an urgent need for research that starts from the requirements of Internet applications and works downward toward network resources, in addition to the predominantly bottom-up work of the networking community. This paper aims to encourage the software-engineering community to participate in this research by providing a starting point and a broad program of research questions and projects.

1 Introduction

In a recent article on the future of software engineering [24], Mary Shaw identified the Internet as both the dominant force for the last 15 years in shaping the software milieu, and the source of software engineering's greatest coming challenges. The challenges center on software that is widely used by—and often created by—the public, in an environment that is ultra-large scale and has no possibility of central control.

Software research has made huge contributions to Internet applications and middleware [10]. These successes have been enabled by an Internet architecture that has provided a relatively stable and universal interface to lower network layers.

Currently, however, Internet architecture is a controversial subject in the networking community. Peoples' ideas are changing fast, and it is increasingly likely that the future Internet will not have an architecture in the sense that it has had one in the past.

The first goal of this paper is to inform software researchers about current trends in the networking community and their likely effect on the future Internet. Sections 2 and 3 summarize the nominal Internet architecture and the reasons it

S. Nanz (ed.), *The Future of Software Engineering*,
DOI 10.1007/978-3-642-15187-3_12,

is inadequate today, both as a characterization of the current Internet and as a basis for meeting projected requirements. Section 4 describes how networking is changing in response to these pressures, and where the current trends are likely to lead.

Sections 3 and 4 show that there is an urgent need for research that starts from the requirements of Internet applications and works downward toward network resources, in addition to the predominantly bottom-up work of the networking community. The second goal of this paper is to encourage the software-engineering community to participate in this research. Section 5 provides a starting point in the form of a well-defined and apparently general architectural unit. Based on this foundation, Section 6 presents a broad program of research questions and projects. The paper concludes that Internet evolution presents exciting and important research challenges that the software-engineering community is best qualified to take up.

2 The "classic" Internet architecture

Although the Internet architecture is not precisely defined, there is broad agreement on the properties and protocols in this section. The Internet is organized into layers, a concept that was brought into operating systems by Dijkstra [9] and then adopted for networking. Usually the Internet is described in five layers, as shown in Figure 1.

The physical layer provides data transmission on various media. A link is a machine-to-machine data connection using a particular medium. Links are partitioned into groups called subnetworks. A machine can belong to (be a link endpoint of) more than one subnetwork, in which case it acts as a gateway between them. The architecture is designed to allow great diversity in the physical and link layers.

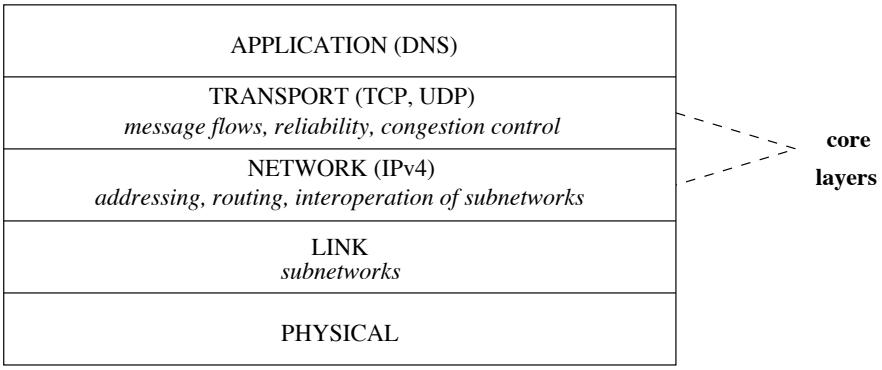

Fig. 1. The five layers of the classic Internet architecture.

The network layer is one of the core Internet layers. It is referred to as the "narrow waist" of the architecture because it is defined by the single Internet Protocol (IP), which runs on all subnetworks. Each machine has a unique 32-bit address. The address space is organized as a location hierarchy, which allows global routing with routing tables of reasonable size. Each machine is either a *host* or a *router*; hosts run applications and serve users, while routers deliver packets. IP provides best-effort packet delivery between the machines.

The transport layer is the other core Internet layer. It is defined primarily by the two transport protocols User Datagram Protocol (UDP) and Transmission Control Protocol (TCP). UDP sends and receives individual packets. TCP implements reliable, FIFO, duplicate-free byte streams (two-way connections). TCP also implements congestion control.

In the application layer, domain names as supported by the Domain Name System (DNS) are available as a global mnemonic name space. Although all other functions are the responsibility of the applications themselves, there are many functions that are common to a variety of applications. These functions are sometimes implemented and deployed as shared or re-usable middleware. The functions include:

- management of application-specific name spaces;
- directories for discovery of resources and services;
- endpoint mobility;
- enhanced inter-process communication facilities such as remote method invocation, transactions, multi-point connections, multicast, publish/subscribe event-based communication, and trusted-broker services;
- security (authentication, access control, encryption, protection from denial-of-service attacks);
- protocol conversion and data reformatting;
- buffering and caching;
- monitoring and billing for quality of service.

The classic Internet architecture was strongly shaped by the "end-to-end" principle [23], which says that in a layered system, functions should be moved upward, as close as possible to the applications that use them. If this principle is followed, applications do not incur the costs and complexity of core functions that they do not use, nor do they contend with core functions that conflict with their goals.

Originally TCP was the only IP transport protocol. In the best-known application of the end-to-end principle, TCP was separated from IP, and UDP was added, when the early designers realized that TCP is ill-suited to real-time traffic such as voice [6]. TCP reliability and ordering introduce delays (while lost packets are retransmitted). Voice protocols cannot tolerate delay, and they deal with lost packets by other means such as interpolation.

A different and equally influential principle that is usually confused with the end-to-end principle is the "fate sharing" principle.[1] The fate sharing principle

[1] The confusion arises because the name "end-to-end" actually fits the fate sharing principle better than it fits its own principle.

says that it is acceptable to lose the state information associated with an entity only if, at the same time, the entity itself is lost [6]. In the best-known application of the fate-sharing principle, all TCP functions run in the hosts at the endpoints of TCP connections. Even if there are transient failures in the network, the hosts can continue to communicate without having to reset their states.

3 The real Internet

The classic Internet architecture is a victim of its own success. Having succeeded so well at empowering users and encouraging innovation, it has been made obsolete by explosive growth in users, traffic, applications, and threats.

3.1 Network management and operations

The Internet was not designed with management in mind, yet the administrators of today's networks face critical problems of configuration, traffic engineering, routing policy, and failure diagnosis. Their tools for understanding network traffic are poor, and their mechanisms for controlling network operations do not offer a predictable relationship between cause and effect.

Most of the Internet consists of autonomous subnetworks that are either operated by organizations for their own purposes, or operated by service providers for profit. In this environment, economic incentives work against many important goals [11]. No network operator has the power to provide better end-to-end quality of service (including bandwidth or latency guarantees, high availability, and fast DNS updates) across the boundaries of autonomous subnetworks. Because no network operator has the power to provide it, no operator can charge for it, and so no operator has an incentive to support it. Furthermore, no operator has a strong incentive to implement improved technology for global network management, because improvements yield no benefit until most other autonomous subnetworks have also implemented them.

Internet routing is beginning to have serious problems of scale. The routing table in a typical router now has 300,000 entries, and these must be stored in the fastest, most expensive types of memory to maintain routing speed.[2] There are efforts to move toward a scheme in which a typical routing table has one entry per autonomous system, which points to a router that can route to all the addresses for which that autonomous system is responsible.

3.2 Middleboxes

To understand the issues of Internet software, it is important to consider *middleboxes*. A *middlebox* is a stateful server that lies in the middle of a communication path between endpoints. It is distinguished from hosts, which are the endpoints

[2] One of the main obstacles to acceptance of IPv6 is the high cost of routers able to handle 128-bit IP addresses.

of communication paths. It is also distinguished from routers, which maintain no state concerning particular instances of communication. A middlebox can perform any function desired [5], in contrast to routers, which are constrained to forwarding packets without altering them.

Middleboxes are ubiquitous in the Internet, as will be discussed in the next two subsections. They have always been controversial, partly because of the unfortunate things that have been done with them, and partly because every middlebox violates the principle of fate sharing.

3.3 Network and transport layers

By the early 1990s two deficiencies of the classic Internet architecture had become recognized as major problems:

- The 32-bit IPv4 address space is too small for the demand.
- There is no provision for secure private subnetworks.

The address space was effectively extended by Network Address Translation (NAT), which allows many machines on a private subnetwork to share a single public IP address. Private subnetworks were fenced off by firewalls. Both are implemented by middleboxes, and NAT and firewall functions are often combined in the same middlebox. These middleboxes are embedded so completely in the network (IP) layer that basic understanding of Internet reachability requires taking them into account [27].

NAT boxes and firewalls are everywhere: all business enterprises and Internet service providers have them—maybe even several tiers of them—and they are packaged with most residential broadband offers. Despite their obvious value, NAT and firewalls do enormous harm from the perspective of those who need to build and deploy applications. This is not because their functions or middleboxes in general are intrinsically harmful, because they are not [26]. The harm is done by heedless conflation of the concerns of network, transport, and application layers.

To begin with NAT, Figure 2 shows a typical NAT box on the border between a private subnetwork and the public Internet. Many different private subnetworks re-use the same "private" portion of the IP address space. This means that some of the nodes in the subnetwork have distinct private addresses but share a small number of public IP addresses, which are the addresses of the subnetwork's NAT boxes.

On its public side the NAT box has an IP address and many ports.[3] A public port encodes a private *address, port* pair, so that a node on the subnetwork can communicate on the open Internet. An example of this encoding is shown in Figure 2. Because there are obviously fewer public ports than private *address, port* pairs, most public ports are allocated dynamically by the NAT box and released as soon as possible for re-use.

[3] There are many variations in the way that NAT boxes and firewalls work. This section describes common and representative schemes.

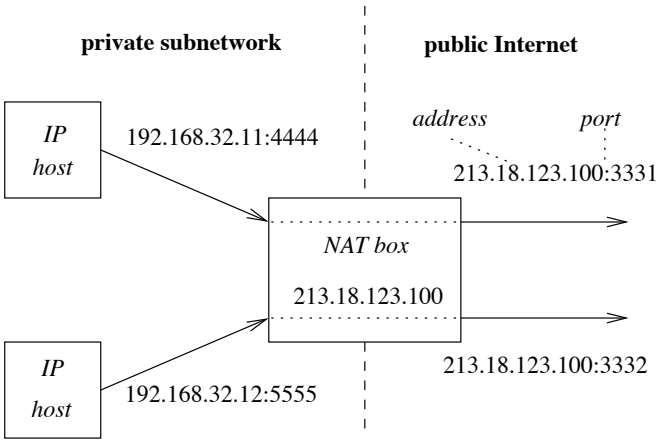

Fig. 2. Source addresses and ports altered by Network Address Translation.

Because of NAT, many machines on private subnetworks do not have persistent public addresses. It is impossible for applications on machines outside the subnetwork to initiate communication with the applications on such a machine, even if this would be useful and firewall security is configured to allow it. In this case, application-layer concerns are sacrificed for network-layer goals.

Port numbers are transport-layer mechanisms used by TCP and UDP to distinguish different sessions or applications at the same node (applications are named by well-known ports). Yet NAT boxes alter port numbers for network-layer reasons. As a result of this conflation of concerns, applications can no longer use them the way they are supposed to be used. For one example, even if a NAT box is configured to leave well-known ports alone, at most one node of the private subnetwork can receive requests at a well-known port. For another example, remote port manipulation by NAT boxes means that an application cannot observe, control, or communicate its own port numbers.

Turning to firewalls, a NAT box often acts as or in conjunction with a firewall. In its default setting, a firewall only allows communication initiated from inside the private subnetwork. A node in the subnetwork sends a message to an IP address and port on the public Internet. Both source and destination *address, port* pairs are stored in the firewall, and only subsequent messages from the public destination to the private source are allowed in.

Peer-to-peer communication can only be established if one endpoint initiates a request and the other endpoint accepts it. Until 2004 it was considered impossible to establish a TCP connection between two peers behind different default firewalls. Even now it only works 85-90% of the time, because the solution is a hack depending on the details of firewall behavior, which vary widely [13].

All of these issues are relatively unimportant for client-server applications such as Web services, at least when the clients are private and the servers are

public. The issues are critical, however, for peer-to-peer applications such as voice-over-IP.

There is a large selection of mechanisms to make peer-to-peer applications such as voice-over-IP possible. However, all of them have major flaws or limitations. NAT boxes and firewalls are sometimes configured manually to allow this traffic, but this is too difficult for most people to do, and violates the security policies of large organizations. Clients, application servers, NAT boxes, and firewalls can all cooperate to solve the problems automatically, but this requires a degree of coordination and trust among software components that is rarely feasible. *Ad hoc* mechanisms such as STUN servers[4] exist, but these mechanisms are fragile, and do not work with common types of NAT, types of firewall, and transport protocols.

Finally, when implementors cannot use TCP because of firewalls, they sometimes re-implement the features of TCP that they want on top of UDP.[5] In addition to the waste of effort, this is almost certain to subvert TCP congestion control.

3.4 Middleware and applications

Applications need a broad range of effective mechanisms for security and endpoint mobility. Although is not clear whether these mechanisms should be in networks, middleware, or applications (or all three), it is clear to everyone that current capabilities are inadequate.

Many of the middleware functions listed in Section 2 cannot be provided in accordance with fate-sharing, or cannot be provided as well [4, 29]. For example, the only way to protect endpoints against denial-of-service attacks is to detect malicious packets before they get to the endpoint. So this function must be provided in the network, even though detecting malicious packets often requires maintaining state information that violates the principle of fate-sharing.

Application middleboxes were more controversial ten years ago. By now the philosophical debate seems to be over, with the trend toward "cloud computing" having finalized the legitimacy of servers in the network. Furthermore, fate sharing is less important now because network servers are commonly run on high-availability clusters.

Although the debate is over, a serious problem remains: the Internet architecture does not support application-level middleboxes. Consider, for example, the problem of introducing Web proxies into the paths of HTTP messages. There are many kinds of Web proxy, providing functions such as caching, filtering, aggregating, load-balancing, anonymizing, and reformatting. They serve the interests of diverse stakeholders such as users, parents, employers, Internet service providers, and Web content providers.

[4] A machine behind a NAT box can query a STUN server to find out what address and port the NAT box has currently assigned to the machine.

[5] Because UDP does not require an initial handshake, it is easier to traverse firewalls with UDP.

Figure 3 shows the desired path of an HTTP request, from a browser on an employee's desk through three proxies:

- a privacy proxy desired by the employee, to reduce information leakage and annoyance;
- a filtering proxy desired by the employing enterprise, to block access to some Web sites;
- a caching proxy desired by the Internet service provider, to improve performance and conserve resources.

The choices for implementing this chain by introducing the middleboxes into the path of the HTTP request are not attractive.

Fig. 3. The path of an HTTP request.

First, a browser or proxy can be configured with the address of the next proxy in the path. This is administratively burdensome and very inflexible. It can also be circumvented by one stakeholder acting against the interest of another stakeholder. For example, why should the privacy proxy (or the Web browser, if there is no privacy proxy) direct requests to the filtering proxy against the user's own interest?

Second, an implementation of IP can look inside messages to find the HTTP requests, and route them in a special way. Finally, a local DNS server can be programmed to answer a query for the IP address of a Web service with the IP address of a local proxy instead. These mechanisms are both messy, burdensome, and inflexible. They break the rules of IP and DNS, respectively, and are only available to some stakeholders.

Because of problems such as these, protocols such as Mobile IP [18] and IP multicast [17] rely on special IP routers, addresses, and routing. In this way their designers avoid the limitations that most application programmers must accept.

3.5 Principles and priorities

The range of problems observed today is not surprising. The Internet was created in simpler times, among a small club of cooperating stakeholders. As the Internet becomes part of more and more aspects of society, it will inevitably be subject to more demands from more stakeholders, and be found deficient in more ways [7].

This final section of the status report turns to impressions of the principles and priorities that have shaped the Internet so far. These impressions are subjective and not completely fair, because the Internet has been shaped by historical,

political, and economic forces beyond anyone's control. All of the tendencies, however, are readily apparent in current networking research as well.

The networking community has a good understanding of the demands of scalability and the value of hierarchy in meeting those demands. I would not say it has a good understanding of the end-to-end principle, because it is usually confused with the principle of fate sharing. Performance, availability, and efficiency are the highest priorities by far.

There are some principles and ideas, understood very well by software engineers, that appear to be unfamiliar to the networking community—or at least to large segments of it:

Complexity matters. The trouble with software is that it can do anything, no matter how complex, convoluted, fragile, incomprehensible, and ill-judged. Software engineers understand the cost of such complexity. Because the networking community underestimates the cost of complexity, it pays no attention to one of the most important problems of the current Internet, which is that it is much too difficult to build, deploy, and maintain networked applications.

Separate concerns. "Tussle" describes the ongoing contention among stakeholders with conflicting interests, as they use the mechanisms available to them within the Internet architecture to push toward their conflicting goals [7]. Because the mechanisms they use are always intertwined with many other mechanisms, and each mechanism was designed to address many concerns, the result of tussle is usually to damage efforts to reach unrelated goals.

With the right abstraction, a structure can be both simple and general. Here the best example is the application protocol SIP [22], which is the dominant protocol for IP-based multimedia applications, including voice-over-IP. SIP is currently defined by 142 standards documents totaling many thousands of pages (and counting) [21]. Each new requirement is met with a new mechanism. There seems to be no conception that a protocol based on better abstractions could be both simpler and more general.

Think compositionally: there is no such thing as a unique element. Networking is full of things that are treated as unique, when they are actually just members of a class that need to be considered in relation to one another and often composed in some way. "The stakeholder deciding on the proxies for an HTTP request" is one such example, as described in Section 3.4, and "a communication endpoint" is another. For example, if a proxy blocks a request for some reason, then the proxy is the endpoint of the communication, rather than the IP host to which it is addressed. Compositional thinking will be discussed further in Section 6.

From the viewpoint of Internet users and application programmers, there are requirements that sometimes equal or exceed performance, availability, and efficiency in priority. These include ease of use, correctness, predictability, and modularity. The use of functional modeling and formal reasoning to help meet such requirements is all-but-unknown in the networking community.

4 Internet trends and evolution

4.1 Trends in practical networking

As we would expect, people have found a variety of ways to work around the shortcomings of the Internet as they perceive them. In addition to code workarounds, two of them operate on a larger scale.

First, many new networks have gateways to the Internet but do not run IP. These include mobile telephone networks,[6] and also many other wireless, enterprise, and delay-tolerant networks.

Second and most important from the perspective of software engineering, enterprises deploy what they need in the form of Internet *overlays*. An *overlay* is a custom-built distributed system that runs on top of the core layers of a network, for the purpose of simulating a different or enhanced architecture. In general an overlay consists of user machines, servers, protocols, data structures, and distributed algorithms. In general an overlay can cross the boundaries of autonomous subnetworks.

Many overlays run on the current Internet. Akamai runs a content-delivery overlay providing high availability, low latency, and fast DNS lookup/update [1]. Skype achieved instant popularity as a voice-over-IP service partly because it was the first such peer-to-peer service to penetrate firewalls successfully. Many global enterprises rely on Virtual Private Networks (VPNs), which are overlays, for security.

Returning to the discussion of voice-over-IP in Section 3.3, Skype uses an overlay solution to the firewall problem as follows [3]. Users on private subnetworks initiate TCP connections to servers or designated peers on the open Internet. These servers connect with each other, and all messages between peers on private subnetworks are relayed by these servers. This is a very inefficient way to transmit high-bandwidth real-time voice data, which should travel between endpoints by the shortest possible path [31].

4.2 Trends in networking research

In the networking community, researchers have been working on deficiencies in the Internet architecture for at least a decade.

In the first half of the 2000s, the research climate was dominated by the belief that research is pointless unless its results can be adopted easily within the existing Internet. Attempting to work within this constraint, people realized that the current architecture makes solving some problems impossible. The idea of a "clean slate" architecture became popular, if only as an intellectual exercise that might lead to fresh ideas and a more scientific foundation for research. Under the title "Next Internet," clean-slate research gained funding and attention worldwide. Researchers gave serious attention to how the results of such work might find their way into practice [12, 19].

[6] The public landline telephone network does not run IP either, but it would not be expected to do so, as it preceded IP by a century.

Even before the question of how results could be used, a clean-slate research agenda faces the question of how new ideas can be evaluated. For an experiment to be convincing, the experimental network must be large-scale and open to public use.

The answer to this question, according to widespread opinion in the networking community, is *virtualization*. This refers to technology for supporting, on a real network, a set of independent virtual networks. Virtualization must guarantee each virtual network a specified slice of the underlying network resources, and must also guarantee that applications and data on different virtual networks are isolated from one another. Virtualization can provide a large-scale test-bed for multiple parallel experiments, at reasonable cost [2]. Virtualization technology itself is now a major subject of research.

There are two ways that these research trends could lead to success. One is that a particular "Next Internet" architecture emerges as the leader and is adopted globally as the Internet architecture for the next few decades. The other is that no architecture emerges as the leader, yet virtualization works so well that no single architecture is required. These are the "purist" and "pluralist" outcomes, respectively [2].

How would a pluralist outcome be different from the current situation, in which more and more stakeholders resort to overlays? As envisioned, the pluralist outcome has three major advantages:

- Provided that virtualization comes with the right economic incentives, a virtual network can get a guaranteed allocation of resources end-to-end, so it can offer predictable quality of service to its users.
- Virtualization is intended to expose the capabilities of the underlying hardware as much as possible, so that virtual networks can monitor and exploit them better than is possible now.
- A virtual network is a clean slate, without the obstructions and complications of existing core layers that current overlays must contend with. In allowing applications to get as close to the physical and link layers as necessary, it is a return to the end-to-end principle.

4.3 Prospects for evolution

Concerning virtualization and "Next Internet" investigations, Roscoe expresses a popular viewpoint in regarding the pluralist outcome, not as a consolation prize, but as the preferred alternative [20]. Certainly it seems the more likely result, given that the purist outcome requires the kind of global consensus that is very difficult to achieve in any public concern.

If neither of these outcomes takes hold, then the Internet will certainly continue evolving in ways that make the classic IP architecture less important. There will be more networks that do not use IP, and more overlays designed to avoid various aspects of it. Overlays will become "application-specific networks" because all networking problems, from application programming to traffic engineering and monitoring, are easier if the application is known.

From a top-down viewpoint, all the possible paths of evolution look similar. There is certainly a new openness to clean-slate thinking and application-specific thinking. Hopefully there is also increased appreciation of the richness of networked applications and the importance of fostering innovation by making them easier to build.

5 A pattern for network architecture

This section defines an *overlay* as a unit of network architecture. In [8], Day argues convincingly that this is the only type of architectural unit. There can be many instances of the overlay type, differing in their ranks, scopes, and properties. The choice of the term "overlay" is discussed in Section 5.4.

Day's pattern for overlays is unusually thorough and explicit. The presentation in Sections 5.1 through 5.2 is a much-abbreviated form of it.

Day's pattern is also well-thought-out and appears to be quite general. For this reason, it seems to be an excellent starting point for further research on network architecture.

5.1 Definition of an overlay

An *overlay* is a distributed inter-process communication facility. It has *members*, each of which is a process running on some operating system on some machine. It also has a *name space* and some mechanism for enrollment (either dynamic or by static configuration) by which processes become members and get unique names from the name space. An overlay process belongs to only one overlay. The membership of an overlay is its *scope*.

An overlay offers a communication service to overlays and applications above that use it. From the perspective of the upper overlay (or "overlay" for short), the programming interface to the lower overlay (or "underlay" for short) gives overlay members the abilities to:

- register and authenticate themselves with the underlay;
- provide information to the underlay about their location, *i.e.*, a member of the underlay on the same machine;
- provide information to the underlay about which overlay members have permission to communicate with them;
- authorize expenditures of resources in the underlay;
- request a communication session with another member of the overlay (by overlay name), and receive an identifier for it;
- accept a requested communication session (associated with the requestor's overlay name) and receive an identifier for it (which need not be the same as the requestor's identifier);
- send or receive messages in a session, using its identifier; and
- close a communication session.

As with enrollment, an overlay can give information to an underlay dynamically or by static configuration.

To implement the sessions of the communication service that it offers, an underlay includes mechanisms to:

- maintain directories with the locations and permissions of registered overlay members;
- enforce the permissions and authorizations of the overlay;
- allocate, manage, and release resources, which often include sessions in other underlays below itself;
- route and forward messages, which may have data units different from those of the overlay;
- perform error control and congestion control on transmission of messages; and
- account for resources used by the overlay.

Returning to a focus on a single generic overlay, the members of an overlay are sometimes partitioned into hosts and routers, but this is not a necessary distinction. The *rank* of an overlay is its place in the use hierarchy.

In Day's view all of the classic IP core is one overlay, including IP, TCP, UDP, and DNS.[7] This is consistent with the nature of current non-IP overlays, which contain all these parts. For example, *i3* is a well-known overlay suitable for a variety of purposes [25].

5.2 Private subnetworks

Figure 4 shows an arrangement of overlays suitable for an application that traverses the boundaries of private networks. A and D are application processes. The application (described as an overlay, although it may not export a service for higher applications) must have at least as many processes as participating user machines. The application has a permission scheme in which A can initiate a session with D. All processes are labeled with their names within their overlay, and registered as co-located with the processes directly below them in the diagram.

Below the application is an overlay for the application infrastructure, containing a process on every machine that participates in the application, whether user or server/router. For the session between A and D, this overlay checks permissions and chooses route **a, b, c, d**.

At the lowest rank of the figure there are three overlays, a public network and two private networks. Processes in these overlays represent points of attachment to these networks. As we can see in the figure, processes **b**, *b*, and *b'* are co-located at a machine that serves as a gateway between public and private networks, as are processes **c**, *c*, and *c'*. The arrows show the path of a message from A to D.

This structure provides opportunities for security that do not depend on discriminating arbitrarily against peer-to-peer applications, as firewalls usually do. Security is discussed further in Section 6.4.

Each network overlay need only have a name space large enough for its members, and it is independent of the name space of any other overlay. Thus the

[7] The distinction between TCP and UDP is de-emphasized.

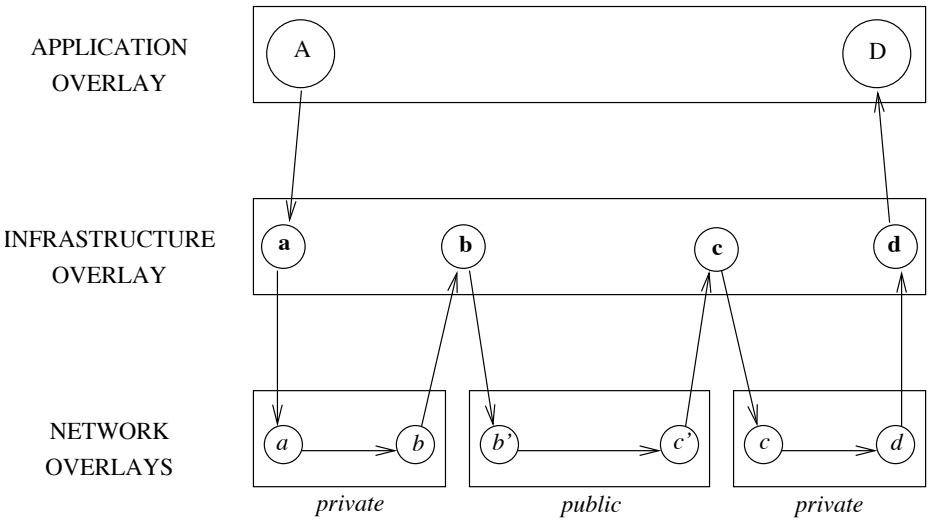

Fig. 4. Interoperating networks are overlays with the same rank.

structure is as good as NAT for managing name spaces of limited size. Application processes refer to each other by names in their own overlay, regardless of where they are located or attached.

5.3 Mobility and multihoming

Mobility is the movement of processes in one overlay with respect to other overlays. Figure 5 is similar to Figure 4, except that the vertical lines show the registered locations of processes. The dashed lines are permanent locations while the dotted lines are transient locations.

Initially **b** is located at *b* in the leftmost network, and **a** can reach **b** in one hop in the infrastructure overlay. Later the machine containing B and **b** moves to the rightmost network, so that it loses its point of attachment *b* and gains a point of attachment *b'*. At this later time, the infrastructure overlay must route messages from **a** to **b** through **c**.

Throughout the movement of this machine, the infrastructure overlay can maintain a session between A and B, because it always has the ability to reach **b** from **a**. This illustrates that the scope of an overlay includes all processes that can reach one another without forwarding in a higher overlay. In contrast, processes *c* and *c'* are not in the same overlay, and cannot reach each other without forwarding in the infrastructure overlay.

Mobility and multihoming are different points on a continuum. With multihoming, a process maintains multiple points of attachment, possibly to different overlays, on a permanent basis. For example, **c** is multihomed to *c* and *c'*. With

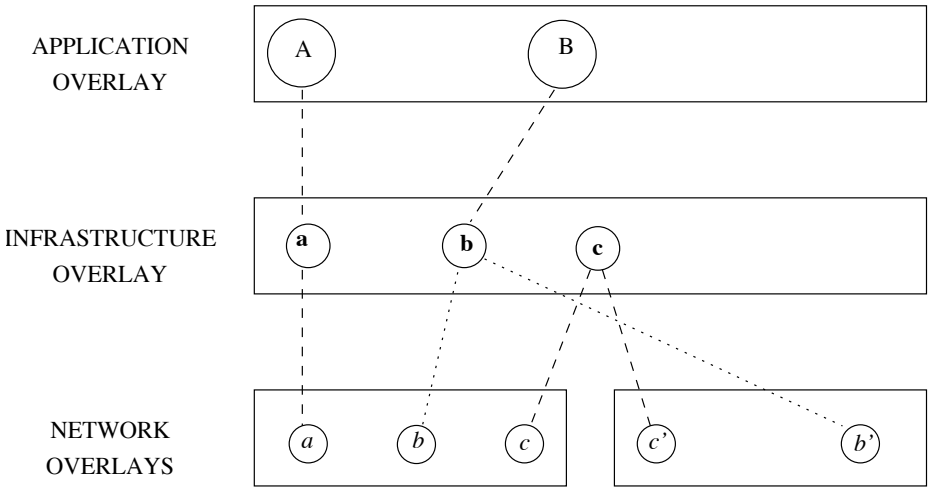

Fig. 5. Mobility is a dynamic relationship between overlays.

mobility, a process may never have two points of attachment at the same time, or have two only transiently.

5.4 Overlay composition

Previous work on overlay composition has proposed two operators for composing overlays, *layering* and *bridging* [16, 15]. *Layering* is the use of one overlay by another overlay of higher rank. *Bridging* is an operator that composes two overlays of the same rank, such as the networks in the bottom ranks of Figures 4 and 5.

As these figures show, bridging is a top-down concept, something that an overlay does with two overlays below it. As an operator joining two overlays of the same rank, bridging is always under-specified and probably a step in the wrong direction. A proper operator would have to specify which are the common elements of bridged overlays, when and how a message is forwarded from one overlay to another, and what service the composed overlays offer to a higher level—all very difficult to do from the bottom up. Such errors are common in networking, due to the chronic absence of specifications (whether formal or rigorous but informal).

I take as a working hypothesis—to be exploited and evaluated—that Day's pattern is the correct unit of architecture. Given its importance, its name is also important. The three candidates are "layer," "overlay," and Day's name Distributed IPC Facility (DIF, where IPC is Inter-Process Communication). "Layer" is not a perfect name because it is too easy to misunderstand it as a rank (as in Figure 4) or as a global structure that all vertical paths must traverse (as in Figure 1, or in cakes). "Overlay" is not a perfect name because of

its implication that it must *lie over* something else, which virtualization aims to make less relevant. Acronyms, and especially recursive ones, impede intuition. Because the confusion caused by "layer" seems much more serious than the confusion caused by "overlay," I have chosen "overlay."

6 Research challenges, from the top down

Internet evolution is creating interest in a large number of research questions concerning network architecture, many of which should be answered from a top-down, application-centric viewpoint rather than a bottom-up, resource-centric viewpoint. There should be no artificial distinction between "middleware" and "network," but rather a well-organized hierarchy of abstractions and implementations, problems and solutions.

6.1 Overlay organization

What is the range of functions and cost/performance trade-offs that overlays can provide? Clearly evolution should move in the direction of offering more and better choices to stakeholders and application developers.

How are overlays arranged in a use hierarchy? Presumably there will be overlays with more application services at the top, and overlays with more clever resource management at the bottom.

In an organization of overlays, how many overlays should there be, and what are their scopes? These would seem to be the most important questions for understanding and achieving scalability.

Within an overlay, can the interests of different stakeholders be represented and composed in an orderly manner? Section 6.3 discusses a particular example of such composition.

As with any mechanism for modularity, layering imposes some overhead in terms of extra processes within an operating system, extra message headers, *etc.* How serious is this overhead? Are there techniques for minimizing it when necessary, and can they be used without destroying valuable boundaries?

An Internet of new overlays would require a tremendous amount of software. To what extent can overlay code be parameterized? How can an overlay be composed from smaller re-usable pieces?

Many of these questions are related to large bodies of research in programming languages and software engineering, far too numerous to mention here. New networking research should be drawing on their successful ideas.

6.2 Overlay interaction

Vertically adjacent overlays should interact only through their interfaces. Is the interface template in Section 5.1 enough, or is more inter-overlay communication required? When are vertically adjacent overlays bound together? Can it be delayed until runtime?

What about the possibility of explicit interaction between overlays that are related vertically but not directly adjacent? For example, could a topmost overlay choose the bottom overlay to be used by a middle overlay between them? Could there or should there be any explicit interface between top and bottom overlays?

How does interaction across the interfaces of the use hierarchy represent and compose the interests of different stakeholders? One issue is already handled explicitly: an overlay can account for and charge for resources used by an upper overlay.

Configuration is part of the interface between layers. In the Internet today, configuration at all levels is a huge headache and major source of errors. Can we design overlays to ease the task of configuration?

At a deeper semantic level, what is the specification of an overlay (the service it offers) and what are its assumptions about the overlays it uses? How can formal reasoning be used to verify that an overlay's assumptions and implementation satisfy its specification, and that the specification of an overlay satisfies the assumptions of an overlay that uses it? Are there principles governing the properties of overlays and how they compose in the use hierarchy?

6.3 Communication primitives

Message transmission (UDP), reliable FIFO connections (TCP), and multicast are familiar Internet primitives. Middleware commonly implements remote method invocation and transactions.

It seems that there might be other communication primitives that could be provided as "sessions" in the overlay pattern. Any of the following possibilities, however, might turn out to be minor variants of other primitives, or so intertwined with an application that they cannot be separated as an application-independent service.

An *abstract name* is a name that does not permanently denote a specific overlay process. Several possible primitives are related to abstract names. For one example, a multicast name denotes a set of processes intended to receive all transmissions in some class. Other examples include publish/subscribe event-based communication, message-oriented anycast, and connection-oriented anycast. (In connection-oriented anycast, a connection request can go to any member of an abstractly named anycast group, but once the connection is formed to a particular process it is point-to-point.) The semantics of abstract names has been explored in [28].

In many high-performance applications today, processes migrate from one machine to another. This is similar to mobility but not identical, as a process changes the machine to which it is attached.[8]

The most interesting possibility in this area is support for applications that are not monolithic, but rather are compositions of several applications. The composed applications can be developed independently, and can represent the

[8] Day views migration as a form of multicast. In his view, a post-migration process is related to, but different from, the pre-migration process.

interests of different stakeholders. For example, Figure 3 can be seen as a composition of four applications (three proxies and a Web server). We know that these applications can be developed independently. As explained in Section 3.4, they represent the interests of four different stakeholders.

In this example the support needed is a way to assemble the servers into the path of the HTTP request that is flexible, convenient, and prioritizes the stakeholder interests in an orderly way. It is not a matter of naming—at least not directly—because the only name given by the user is the name of the Web server at the end of the chain.

This problem is actually a special case of the problem solved by Distributed Feature Composition (DFC) [14, 30]. DFC is an architecture for telecommunication services composed of independent applications. In DFC the analog to Figure 3 is a dynamically assembled, multi-party, multi-channel graph that can change while connected to a particular endpoint, even splitting into two graphs or joining with another graph. The applications appear in the graph because endpoint names and abstract names *subscribe* to them.

Not surprisingly, routing to assemble a DFC graph uses subscriber data and an extra level of indirection, so its overhead is too high for many Internet applications. The challenge here is to find weaker versions that scale better, so that there are adequate solutions for each situation.

6.4 Security

As with the other research topics, the foundation of security is the association between network structures and the stakeholders who control them, as well as the trust relationships among stakeholders. Security mechanisms can relocate trust, but not create it.

Security is often partitioned into three problems: availability, *integrity, i.e.,* controlling changes to the system state, and *confidentiality, i.e.,* keeping information secret.

Two kinds of security are obviously inherent in the overlay pattern. On behalf of its users, an overlay must enforce their policies on which user processes have the authority to change configured state, and which user processes have permission to communicate with which other user processes. On its own behalf, it must enforce its own policies on which applications, overlays, and overlay members can use it, and how they are held accountable for their expenditures. These mechanisms would seem to generalize well to many aspects of availability and integrity.

Confidentiality is most often enforced by encryption. The most natural place for encrypting messages would seem to be within the overlay that is sending and receiving them, rather than in the underlay that is transmitting them.

Theoretical considerations aside, the main point is that buggy software makes bad security. With respect to security, the purpose of network architecture is to make the operation of network software correct, comprehensible, and modular, so that effective security mechanisms can be designed for it.

7 Conclusions

Because of opportunities created by Internet evolution, there are exciting and important research challenges concerning how networks should be designed to meet the growing needs of global networked applications. The observations in Section 3.5 show that software engineers should be participating in this research, because the skills, perspective, and values of our community are much needed.

"Internet architecture," whether in the classic form of Section 2 or the reality of Section 3, has always served as a boundary between the disciplines of networking (below) and distributed software systems (above) [20]. To participate in Internet research, we must break down this boundary. We must gain more familiarity with the design issues of layers below middleware. We must also work harder to bring our perspective to the networking community, whether in industry, academia, or the Internet Engineering Task Force (IETF).

Finally, software engineers should collaborate with researchers in the networking community to test their ideas and find the most efficient ways to implement them. One of the great strengths of the networking community is building efficient, large-scale implementations of distributed algorithms. Another great strength of the networking community is experimental evaluation of ideas. And only collaborative research will foster the common terminology and shared values needed for both communities to affect technology in practice.

Acknowledgment

Most of my understanding of Internet evolution is due to the generous and expert guidance of Jennifer Rexford.

References

1. Afergan, M., Wein, J., LaMeyer, A.: Experience with some principles for building an Internet-scale reliable system. In: Second Workshop on Real, Large Distributed Systems (WORLDS '05). USENIX Association (2005)
2. Anderson, T., Peterson, L., Shenker, S., Turner, J.: Overcoming the Internet impasse through virtualization. IEEE Computer 38(4), 34–41 (April 2005)
3. Baset, S.A., Schulzrinne, H.G.: An analysis of the Skype peer-to-peer Internet telephony protocol. In: Proceedings of the 25th Conference on Computer Communications (INFOCOM). IEEE (April 2006)
4. Blumenthal, M.S., Clark, D.D.: Rethinking the design of the Internet: The end-to-end arguments vs. the brave new world. ACM Transactions on Internet Technology 1(1), 70–109 (August 2001)
5. Carpenter, B., Brim, S.: Middleboxes: Taxonomy and issues. IETF Network Working Group Request for Comments 3234 (2002)
6. Clark, D.D.: The design philosophy of the DARPA Internet protocols. In: Proceedings of SIGCOMM. ACM (August 1988)
7. Clark, D.D., Wroclawski, J., Sollins, K.R., Braden, R.: Tussle in cyberspace: Defining tomorrow's Internet. IEEE/ACM Transactions on Networking 13(3), 462–475 (June 2005)

8. Day, J.: Patterns in Network Architecture: A Return to Fundamentals. Prentice Hall (2008)
9. Dijkstra, E.W.: The structure of the 'THE' multiprogramming system. Communications of the ACM 11(5), 341–346 (May 1968)
10. Emmerich, W., Aoyama, M., Sventek, J.: The impact of research on the development of middleware technology. ACM Transactions on Software Engineering and Methodology 17(4), Article 19 (August 2008)
11. Feamster, N., Gao, L., Rexford, J.: How to lease the Internet in your spare time. Computer Communications Review 37(1), 61–64 (January 2007)
12. Feldmann, A.: Internet clean-slate design: What and why? Computer Communications Review 37(3), 59–64 (July 2007)
13. Guha, S., Francis, P.: An end-middle-end approach to connection establishment. In: Proceedings of SIGCOMM. pp. 193–204. ACM (August 2007)
14. Jackson, M., Zave, P.: Distributed Feature Composition: A virtual architecture for telecommunications services. IEEE Transactions on Software Engineering 24(10), 831–847 (October 1998)
15. Joseph, D., Kannan, J., Kubota, A., Lakshminarayanan, K., Stoica, I., Wehrle, K.: OCALA: An architecture for supporting legacy applications over overlays. In: Proceedings of the Third USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '06) (2006)
16. Mao, Y., Loo, B.T., Ives, Z., Smith, J.M.: MOSAIC: Unified declarative platform for dynamic overlay composition. In: Proceedings of the Fourth Conference on Future Networking Technologies. ACM SIGCOMM (2008)
17. Mysore, J., Bharghavan, V.: A new multicasting-based architecture for Internet host mobility. In: Proceedings of the Third Annual ACM/IEEE International conference on Mobile Computing and Networking. ACM (1997)
18. Perkins, C.E.: Mobile IP. IEEE Communications (May 1997)
19. Ratnasamy, S., Shenker, S., McCanne, S.: Towards an evolvable Internet architecture. In: Proceedings of SIGCOMM. ACM (August 2005)
20. Roscoe, T.: The end of Internet architecture. In: Proceedings of the Fifth Workshop on Hot Topics in Networks (2006)
21. Rosenberg, J.: A hitchhiker's guide to the Session Initiation Protocol (SIP). IETF Network Working Group Request for Comments 5411 (2009)
22. Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., Schooler, E.: SIP: Session Initiation Protocol. IETF Network Working Group Request for Comments 3261 (2002)
23. Saltzer, J., Reed, D., Clark, D.D.: End-to-end arguments in system design. ACM Transactions on Computer Systems 2(4), 277–288 (November 1984)
24. Shaw, M.: Continuing prospects for an engineering discipline of software. IEEE Software 26(6), 64–67 (November 2009)
25. Stoica, I., Adkins, D., Zhuang, S., Shenker, S., Surana, S.: Internet indirection infrastructure. In: Proceedings of SIGCOMM. ACM (August 2002)
26. Walfish, M., Stribling, J., Krohn, M., Balakrishnan, H., Morris, R., Shenker, S.: Middleboxes no longer considered harmful. In: Proceedings of the Sixth Usenix Symposium on Operating Systems Design and Implementation. ACM (December 2004)
27. Xie, G., Zhan, J., Maltz, D.A., Zhang, H., Greenberg, A., Hjalmtysson, G., Rexford, J.: On static reachability analysis of IP networks. In: Proceedings of IEEE Infocom. IEEE (March 2005)
28. Zave, P.: Address translation in telecommunication features. ACM Transactions on Software Engineering and Methodology 13(1), 1–36 (January 2004)

29. Zave, P.: Requirements for routing in the application layer. In: Proceedings of the Ninth IEEE International Conference on Coordination Models and Languages. pp. 19–36. Springer-Verlag LNCS 4467 (2007)
30. Zave, P.: Modularity in Distributed Feature Composition. In: Nuseibeh, B., Zave, P. (eds.) Software Requirements and Design: The Work of Michael Jackson. Good Friends Publishing (2010)
31. Zave, P., Cheung, E.: Compositional control of IP media. IEEE Transactions on Software Engineering 35(1) (January/February 2009)

Mining Specifications: A Roadmap

Andreas Zeller

Saarland University, Saarbrücken, Germany
zeller@cs.uni-saarland.de
WWW home page: http://www.st.cs.uni-saarland.de/zeller/

Abstract. Recent advances in software validation and verification make it possible to widely automate whether a specification is satisfied. This progress is hampered, though, by the persistent difficulty of *writing* specifications. Are we facing a "specification crisis"? In this paper, I show how to alleviate the burden of writing specifications by reusing and extending specifications as *mined* from existing software and give an overview on the state of the art in specification mining, its origins, and its potential.

Keywords: formal methods, software verification and testing, specifications, mining

1 Introduction

Any systematic engineering activity requires a *specification*—a description of the goals that are to be achieved. Software construction requires a specification of what is to be built. Software quality assurance requires a specification to be compared against. Systematic specification is at the heart of all engineering; as Brian Kernighan once famously quoted "Without specification, there are no bugs—only surprises."

But where do we get the specifications from? Textbooks on Software Engineering tell us that specifications are to be elicited at the beginning of a project, as part of requirements engineering. This phase produces a hopefully complete and consistent set of requirements, forming the base for all future work. But these requirements are usually *informal,* leaving room for ambiguity, thus making it hard to validate them automatically—and again, thus substitute bugs by surprises.

Formal methods, of course, promise to address all these issues; and indeed, the past decade has made tremendous advances in formal verification techniques. Indeed, *checking* specifications is now automated to a large degree, with techniques like software model checking, static program analysis, or automated test case generation. All of these techniques, however, validate properties that are easy to formulate: "There shall be no null pointer dereference.", "There must be no memory leak", or plain and simple: "Thou shalt not crash." In contrast to checking specifications, *writing* specifications has made little advances, though. Twenty-five years after its conception, *design by contract* as embodied into the

S. Nanz (ed.), *The Future of Software Engineering*,
DOI 10.1007/978-3-642-15187-3_13,

Eiffel programming language is still the only one convincing contender integrating specification and programming into a single, seamless, and easy-to-use package. Still, the burden of specification remains.

Much of this burden simply comes from the burden of *decisions* as they have to be made in any software project sooner or later. A significant burden, however, also comes form the fact that existing programs and processes so far lack formal specifications. Using a library, implementing a protocol, porting a business process—every time, we have to understand and infer what is happening or what is supposed to happen; every time, we need *specifications of existing systems* such that one can validate new systems as a whole.

In 1968, the term "software crisis" denoted the inability of software engineers to catch up with the rapid increase in computing power and the complexity of the problems that could be tackled. Today, we have a similar issue: Our abilities to generate, verify, and validate software have increased tremendously. Our abilities to write specifications have not. Are we facing a "specification crisis"?

One way to alleviate the burden of writing specifications is to *reuse* and *extend* existing specifications. The obvious source for these is *existing software.* After more than 40 years of large-scale programming, our existing systems encode decades of knowledge and experience about the real world. While we can safely rely on having production systems and source code available, the existence of any other artifacts cannot be taken for granted. In particular, we cannot assume having an abstract description of an existing system in a form that would be amenable to automated processing. Therefore, we must go and *extract* such specifications from existing systems.

This extraction is the purpose of *specification mining*—given an existing piece of software, to extract a specification that characterizes its behavior. To be useful, such a mined specification must satisfy two requirements:

1. The specification must be as *accurate* as possible, including possible behavior and excluding impossible behavior.
2. The specification must be *efficient*, abstracting from details while still capturing essential behavior.

These two requirements are in conflict with each other. Simply said, one cannot have complete accuracy and perfect abstraction at the same time. Furthermore, there may be multiple abstractions to be mined, all of which may or may not be relevant for the task at hand. And what is it that makes one abstraction more effective than another one? Today, we do not have answers to these issues. Yet, the combination of previously separate techniques such as static analysis, dynamic analysis, software mining, and test case generation brings new promises for mining specifications that are accurate *and* efficient enough to promote wide-scale usage. In this paper, I give an overview on the state of the art in specification mining, its origins, and its potential.

Concepts

- ✏ While checking specifications has made tremendous progress, writing specifications is still a burden
- ✏ Existing systems encode decades of knowledge and experience about the real world
- ✏ We should strive and mine accurate and efficient specifications from these existing systems

2 The Problem

As a motivating example, consider Figure 1, showing the source of the Eiffel `sqrt()` method. The Eiffel language integrates preconditions (`require`) and postconditions (`ensure`), specifying precisely what `sqrt()` requires and what it produces. Based on this specification, one can use `sqrt()` without considering its implementation—which is not that interesting, anyway, as it simply refers to some internal C function.

But let us now assume we *only* have the implementation and want to derive its pre- and postconditions. How would we do that? Immediately, we are faced with a number of challenges:

Inferring postconditions. How can we find out what `sqrt()` does? *Static analysis* such as abstract interpretation or symbolic evaluation give us means to infer and abstract its behavior from the implementation; however, the absence of preconditions (and, in this particular case, the absence of accessible source code) pose severe practical problems; generally, the conservative approximation of static analysis will lead to a superset of actual behavior. An alternative is to observe the *dynamic* behavior in actual runs, which will induce a subset of the possible behavior.

Inferring preconditions. To infer the precondition of `sqrt()`, we either deduce it from its postcondition (which, on top of static imprecision, brings us

```
1 sqrt (x: REAL, eps: REAL): REAL is
2     -- Square root of x with precision eps
3 require
4     x >= 0: eps > 0
5 external
6     csqrt (x: REAL, eps: REAL): REAL
7 do
8     Result := csqrt (x, eps)
9 ensure
10    abs (Result ^ 2 - x) <= eps
11 end -- sqrt
```

Fig. 1. A square root function in Eiffel [5]

a chicken-and-egg problem). An alternative is to statically or dynamically examine a number of legal *usage examples* which will induce a subset of possible usages.

Inferring interaction. Specifications not only come in the form of pre- and postconditions; they may also express *sequences* of events; for instance, a number of API routines may only be used in a specific order. Deducing possible combinations of routines statically is even harder than deducing the specification for a single routine alone; again, *usage examples* are the best source.

Expressing specifications. The actual postcondition of `sqrt()` uses other methods auch as `abs()` to express the behavior. `abs(x)` simply returns the absolute of `x`. Just like `sqrt()`, the `abs(x)` routine is part of the mathematical run-time library, and therefore, it is appropriate to include it in the *vocabulary* out of which we compose our specifications. For a well-established domain like mathematics, this is less of an issue; but for programs in a specific domain, one will have to rely on provided or mined vocabularies to express conditions.

Filtering specifications. Any specification inference technique has the choice between several ways to express specification, which implies that we need to choose the best. "Best" not only means the most accurate; it also means the most readable, the most expressive, and the most useful for its intended purpose. Most of these features are intrinsically vague, and much work will be needed to refine them.

To make the challenge even harder, we'd like our approaches to be as *automatic* as possible. If we require programmers to annotate their code with axioms or summaries, we may just as well ask them to annotate their code with specifications. And while Bertrand Meyer might say that this would be a good idea in the first place, anyway—let's still try to face the challenge and *mine good specifications.*

Concepts

- ✏ Inferring postconditions is either a static overapproximation or a dynamic underapproximation
- ✏ Inferring preconditions requires real usage examples
- ✏ Specifications need an appropriate vocabulary

3 Mining Programs

The pioneering work in the field first concentrated on mining specifications from *program executions.* Ernst et al. developed the DAIKON tool which automatically abstracts *pre- and postconditions* from program executions [2], providing a specification for each function. Let us assume DAIKON observes the following executions:

```
sqrt( 4.0, 0.0) = 2.0
sqrt( 9.0, 0.0) = 3.0
sqrt(16.0, 0.0) = 4.0
```

On these executions, DAIKON applies a number of *patterns* instantiated with local variables and values. The pattern `$1 * $2 = $3`, instantiated to `x * x = Result` matches all three runs, and therefore is retained as a postcondition; indeed it characterizes well the semantics of `sqrt()`, as long as `eps = 0.0` holds. This condition, together with `4.0 <= x <= 16.0`, is retained as precondition. In this simple example, we already see the main weakness of dynamic invariants: The inferred specifications only match *observed* runs, not all runs. Unless we observe an execution with `eps` $\neq$ `0.0` or a wider value range of `x`, we won't be able to come up with a more precise precondition; and unless we extend our pattern library appropriately, we won't be able to find a concise description of the postcondition. With its pattern library, the DAIKON approach thus provides an *over-approximation* on the observed runs, which again are an *under-approximation* of the possible executions.

Still, the DAIKON approach is the leading one; not only because it was the very first to mine specifications, but also because it is among the few *applicable* ones. The alternative to the under-approximation as learned from executions is the over-approximation as learned from actual *program implementations*. In principle, one could use *symbolic execution* to analyze the `sqrt()` implementation and return the true and accurate postcondition. In practice, we would find numerous problems:

- First, applicability: the implementation may be obscure (as `csqrt()` in our above example, which ultimately maps to appropriate processor instructions).
- Second, complexity: we'd need a full-fledged theorem prover, loaded with all the required mathematical theory, to come up with the proper postcondition.
- Third, the problem is undecidable in general: Any serious static analysis will always be bounded by the halting problem.

The consequence of all this is *conservative over-approximation,* where the enumeration of all that *may* happen can become so long that it is easier to state what *cannot* happen.

Of course, if we *had* a specification for `csqrt()`, then statically deducing the specification of `sqrt()` would be a piece of cake. But this is precisely the point of this paper: Once we have specifications for callees, inferring the specification for the callers is much easier. It is this huge bottom-up approach that we have to face, from `csqrt()` to `sqrt()` to all its clients—and their clients as well.

Concepts

- ✏ From a set of executions, we can learn specifications that match this very set
- ✏ Deducing specifications statically still imposes challenges for practical programs

```
static int find_file (...)
{
    DIR *dirp;
    struct dirent *dirinfo = opendir(".")
    while ((dirinfo = readdir(dirp)) != NULL) {
        ...
    }
    rewinddir(dirp);
    return 1;
}
```

Fig. 2. Skeleton of a defect in cksfv-1.3.13 (from [3])

4 Learning from Usage

To make specifications more accurate, we need more information—and as the information we can extract from the routine itself is limited, we must turn to its *clients*. Indeed, the specification of a routine is not only explicitly encoded in the routine itself, but also *implicitly* in the way it is used:

Static analysis. If we find that `sqrt(x)` is always called with `x` being positive or zero, we may turn this into a precondition for `sqrt()`, considerably narrowing the deduction burden for the remainder of the static analysis. While such findings are hard to deduce statically, usage examples can give us implicit *assumptions* from which real pre- and postconditions can be inferred.

Dynamic analysis. If we find that `sqrt()` is called with differing values of `eps`, we may get rid of the precondition `eps = 0.0` as erroneously inferred by DAIKON. Generally speaking, the more aspects of the behavior we can cover, the more accurate the resulting specification will be.

Such improvements are not only hypothetical. Very recently, a number of experiments have shown how to *leverage* usage examples to make specifications more accurate. Both approaches mine *finite state automata* that describe correct sequences of API invocations.

Leveraging usage examples. Using a lightweight parser, Gruska et al. have been able to mine usage specifications from the 200 million lines of code of the Gentoo Linux distribution [3]. Their specification model is geared towards detecting *anomalies*, i.e. unusual API usage. As an example of an anomaly discovered by their tool, consider Figure 2. Here, a directory of files is processed with `opendir()` and `readdir()`. However, the last call is `rewinddir()` rather than `closedir()` as one would expect, leading to memory leaks and long-term resource depletion. Interestingly, to mine the fact that directory accesses usually end in `closedir()`, one needs to learn from dozens of projects—because a single project rarely has more than one directory interaction. Once learned, though, the resulting specification can be used not only in simple anomaly detection, but for all kinds of further validation and verification purposes.

Leveraging additional runs. For DAIKON, we have seen the fundamental restriction that it can only infer specifications for *observed* runs. Recent advances in test case generation show how to overcome this problem. One could go and generate *random runs* to improve specifications. Or one could go and systematically explore conditions yet uncovered by the specification. This is what the TAUTOKO tool [1] does. It first mines finte-state automata describing API usage in observed executions. It then *enriches* these finite-state automata by generating test cases for situations yet unobserved: "So far, I have always seen `closedir()` as the last action on a `dirinfo` type. What happens if I invoke `closedir()` twice in a row?" The additional knowledge frequently describes exceptional behavior ("You cannot call `closedir()` twice") that is rarely observed in normal executions.

Briefly summarized, additional usage examples will give us more precise preconditions; while additional executions will give us a richer picture of the program's behavior. Both approaches are based on induction; hence, the more instances to learn from, the higher the accuracy.

Concepts

- ✏ Usage examples express assumptions on how the software is used—in particular preconditions
- ✏ When learning from executions, additional runs (and thus usage examples) can be generated on demand
- ✏ This leads to increased accuracy in mined specifications

5 A Hierarchy of Abstractions

As stated in the introduction, a specification must be *efficient*: It must abstract from details while still capturing essential behavior. To abstract from details, it first and foremost needs an expressive *vocabulary*—that is, idioms that express central concepts in little space. *Mathematics* provides a universal vocabulary that allows us to easily express mathematical properties. We can simply use a concept such as "square root" without having to provide the exact definition; specification languages such as Z [4] make extensive use of the vocabulary of mathematics.

But if our program is situated in a specific domain, we also want the idioms of that domain to be used. The behavior of a program in the networking domain will require idioms such as connections, routes, protocols, or state. A banking application will require idioms such as accounts, transactions, debit, or credit. To generate efficient specifications, we shall need domain-specific idioms. But where would we get these idioms from?

Again, the programs themselves may well provide the idioms we are searching for—at least, as long as they follow basics of object-oriented programming. Essentially, we can separate routines into two groups:

- *Query routines* (also called *observers* or *inspectors*) return some aspect of an object's state.
- *Command routines* (also called *mutators*) alter the object's state.

With this separation, established by Bertrand Meyer, we can express the pre- and postconditions of commands by means of *queries*—thus choosing the same abstraction level for the specification as the functionality to be specified. If the implementation lacks appropriate queries, it may be beneficial to implement them as additional functionality—because other clients may profit from them.

While the usage of appropriate idioms will lead to readable, easy-to-understand specifications, they do not address the second problem: For every set of concrete values, there is an infinite number of matching clauses—or in our context, an infinite set of potential specifications. Again, we need efficiency: We want specifications that are as short and concise as possible, yet cover the relevant behavior. But what is the "relevant" behavior? And how can we decide that one specification is more relevant than another?

To start with, not all observed behavior has the same relevance. Behavior that is never checked by any client should not show up in a specification unless we really want this level of detail. What matters most is what *most of the clients depend upon.* Assume we want to mine the specification of a data/time package. Basic local date and time management impacts more clients than non-local time zone management, historic calendars, or non-Gregorian calendars; therefore, any mined specification should start with these most relevant properties first, and, if needed, incrementally proceed towards more and more details. Relevance is in the eye of the beholder; applied to program properties, relevance is in their usage.

At one given abstraction level, though, it is the well-known principle of *Occam's Razor* that should be our guide: we should always aim for the theory with the fewest new assumptions. Assuming that programmers strive for generality, abstraction, and elegance in their implementations, we should strive for the same properties in our mined specifications. With a bit of luck, we may turn lots of implicit assumptions into explicit elegance.

Concepts

- ✏ Efficient specifications must express behavior in terms of *queries*, returning some aspect of an object's state
- ✏ Efficient specifications should be organized *incrementally,* starting with the most relevant behavior
- ✏ The most relevant behavior is the one most clients depend upon

6 Conclusion and Consequences

Mining specifications is still in its very infancy. Coming up with a specification that is both accurate and efficient, covers all the significant behavior and yet manages to abstract away from less relevant details, remains a challenge that

will need the best in program analysis, symbolic reasoning, test case generation and software reverse engineering.

The potential benefits make all these challenges worthwhile, though. Let us assume for a moment that we would actually be able to extract abstract, readable, elegant specifications at the moment programs are written.

- We would see a boost for automatic verification techniques, as all of a sudden, lots of functional specifications at all abstraction levels would become available.
- Understanding legacy programs would be dramatically simplified, as abstract specifications cover the most relevant behavior.
- Programmers no longer would have to write formal specifications; instead, it would suffice to manually validate the extracted abstract behavior against the intended behavior.
- Changes to programs could be immediately assessed in their impact on the abstract specifications.
- We would see completely new styles of implementations and specifications evolving together, with specifications extracted from implementations, and implementations synthesized from specifications.
- In the long run, this means that we will be able to specify, program, and verify on all abstraction levels, starting at the highest (and simplest), and gradually progressing towards more and more details.

Intertwining specification and programming, being able to reason about programs at all levels, will raise software development to new levels, both in terms of quality and productivity. Speaking of research communities, it will also bridge the gap between formal methods and large-scale software engineering, providing new challenges and approaches for either side. And why am I so confident that this will work? Because the integration of specification and implementation has worked so well in the past, as demonstrated in the lifetime achievements of our veneered pioneer, as honored in this very volume. Eiffel has paved the way—now let's go and Eiffelize the world!

Concepts

- ✏ Intertwining specification and programming offers several opportunities for formal methods, software validation, and program understanding
- ✏ This can raise quality and productivity all across software development
- ✏ We can bridge the gap between formal methods and real-world software

For information about the research topics covered in this work, visit the author's research group home page at

http://www.st.cs.uni-saarland.de/

References

1. Dallmeier, V., Knopp, N., Mallon, C., Hack, S., Zeller, A.: Generating test cases for specification mining. In: ISSTA '10: Proceedings of the 2010 International Conference on Software Testing and Analysis. pp. 65–96. ACM, New York, NY, USA (July 2010)
2. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. In: ICSE '99: Proceedings of the 21st international conference on Software engineering. pp. 213–224. ACM, New York, NY, USA (1999)
3. Gruska, N., Wasylkowski, A., Zeller, A.: Learning from 6,000 projects: Lightweight cross-project anomaly detection. In: ISSTA '10: Proceedings of the 2010 International Conference on Software Testing and Analysis. pp. 119–130. ACM, New York, NY, USA (July 2010)
4. Jacky, J.: The way of Z: practical programming with formal methods. Cambridge University Press, New York, NY, USA (1996)
5. Meyer, B.: Eiffel: An introduction. In: Florentin, J. (ed.) Object-Oriented Programming Systems, Tools and Applications, pp. 149–182. Chapman & Hall (1991)

Greetings to Bertrand on the Occasion of his Sixtieth Birthday

Tony Hoare

Microsoft Research, Cambridge

I first met Bertrand in July 1978, at a lovely French Château in le Breau sans Nappe. It provided most luxurious accommodation in the lovely French countryside for two weeks for my family with three children. The food was also superb.

The Château was run by Électricité de France as an educational institution for residential courses. Bertrand was at that time employed by EDF, and organised a Summer School on Computing. It was superbly organised. I learnt there that it is always rewarding to accept an invitation to an event organised by Bertrand; and my latest experience was in Elba as a lecturer in a LASER Summer School a few years ago. Thank you again Bertrand.

At le Breau, Bertrand invited me to lecture on concurrency (Communicating Sequential Processes). Barry Boehm lectured on Software Engineering Economics, and Jean-Raymond Abrial (jointly with Bertrand) on specification in an early version of Z. Of course, since then they have all become famous; and like me, they have all remained active to this day. I have met them again many times in the last thirty years, and they have remained good friends to me and my family.

In 1983 Bertrand migrated to an academic position in California, and in 1985 he founded a Company in California, to implement and market his successful programming language Eiffel. I became editor of the 'Red and White' Prentice Hall International Series in Computer Science. I wanted to persuade Bertrand to write a book for the series. I had no difficulty in doing that. In fact in his reply he said he was already engaged in writing six books, and which one did I want? So my main task was to help him to select a topic on the theory of programming languages, and then to persuade him to stop writing the other five books.

I did not succeed. The agreed and scheduled date in the publisher's contract receded into the past, and Bertrand became ever more deeply apologetic. Finally, he offered a consolation for the delay, in the form of a manuscript for a completely different book, entitled Object-Oriented Software Construction. Would we like to publish that first? Certainly we would. I could see it was an excellent book, and Prentice Hall was surely consoled by the fact that it turned out to be the best-seller of the whole series. It was published in 1988, and was translated to seven languages (many of which, incidentally, Bertrand speaks well). It is still selling very well in its second edition (1997). The original theoretical book on programming (also very good) appeared two years later.

In 1999, I retired from my academic career and moved to Microsoft Research at Cambridge. In January 2001, I visited the Microsoft Headquarters in Redmond, to conduct a survey of the ways in which Microsoft program developers

S. Nanz (ed.), *The Future of Software Engineering*,
DOI 10.1007/978-3-642-15187-3,

were using assertion macros in their code. It was mostly as test oracles, to help discover errors. Bertrand had been invited to Redmond at the same time to talk about Eiffel, the first language which supported the use of assertions as contracts between the writers of separate modules of a large system. In the evening of Saturday 20 January we met at my hotel to reminisce about the past and speculate for the future.

My hope was that Microsoft developers would extend their use of assertions to help discover errors in testing, and would discover later that the assertions would help them to avoid the errors altogether. Bertrand's hope was that Microsoft would adopt the Eiffel philosophy of use of assertions as contracts. I am glad to say that Bertrand's hopes have already been realised. Contracts are available in the languages supported by Microsoft's Visual Studio. My hopes are still hopes: but they are being actively pursued inside Microsoft. There are over a hundred researchers in Microsoft Research world-wide, developing theories, constructing tools and conducting experiments in program verification. There are even more in the development division, actively transferring the technology and improving and adapting the tools for use by different teams.

Bertrand shares with me the long-term ideal that programs should be specified and designed in such a way that the correctness of the delivered code can be reliably checked by computer, using mathematical proof. We felt that this was a challenge in which academic research was needed to make significant progress. I and Jay Misra therefore suggested an IFIP Working Conference on Verified Software, Theories, Tools and Experiments, which Bertrand volunteered to organise – after which its success was assured! This was the start of a series of workshops and special interest groups convening at conferences held throughout the world. Two further full international Conferences have been dedicated to this topic, in Toronto in 2008 and in Edinburgh this year.

Bertrand's enormous contribution to this international endeavour has been derived from his unique experience of the Eiffel project. In pursuit of the long-term scientific ideal of correctness, and exploiting his superb engineering judgment, he has been the chief architect of the EiffelStudio program development environment. This has recruited a community of enthusiastic users, including large institutions and companies. The Eiffel language has continued to develop towards its ideals, and its community has been willing to act as early adopters in experimental use of the new features. This provides a body of experimental evidence which shows the advantage of a principled approach to software engineering. I have reason to be personally grateful for the many discussions that we have held on theory and language design. And we all have good reason to wish him a happy birthday, as well as a long continuation of his extraordinary range of interests and activities.

Yours,

Tony.

Author Index