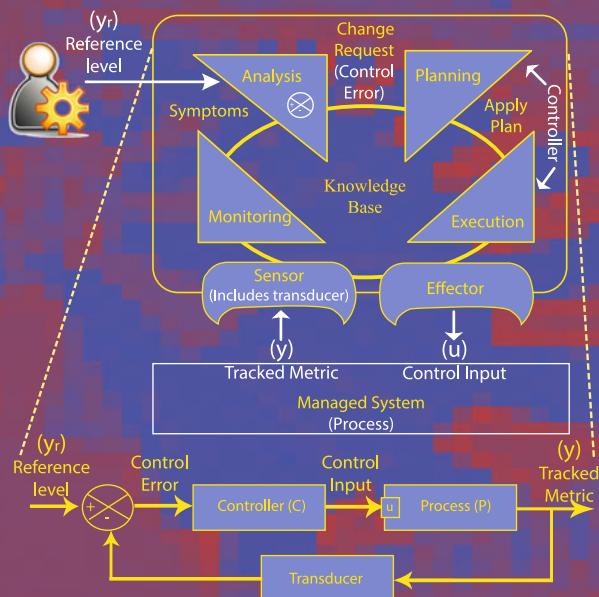


Software Engineering for Self-Adaptive Systems III

Assurances

International Seminar
Dagstuhl Castle, Germany, December 15–19, 2013
Revised Selected and Invited Papers



Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, Lancaster, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Friedemann Mattern

ETH Zurich, Zurich, Switzerland

John C. Mitchell

Stanford University, Stanford, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

TU Dortmund University, Dortmund, Germany

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max Planck Institute for Informatics, Saarbrücken, Germany

More information about this series at <http://www.springer.com/series/7408>

Rogério de Lemos · David Garlan
Carlo Ghezzi · Holger Giese (Eds.)

Software Engineering for Self-Adaptive Systems III Assurances

International Seminar
Dagstuhl Castle, Germany, December 15–19, 2013
Revised Selected and Invited Papers



Springer

Editors

Rogério de Lemos
University of Kent
Canterbury
UK

David Garlan
Carnegie Mellon University
Pittsburgh, PA
USA

Carlo Ghezzi
Politecnico di Milano
Milan
Italy

Holger Giese
Hasso Plattner Institute for Software
Systems Engineering
Potsdam
Germany

ISSN 0302-9743

ISSN 1611-3349 (electronic)

Lecture Notes in Computer Science

ISBN 978-3-319-74182-6

ISBN 978-3-319-74183-3 (eBook)

<https://doi.org/10.1007/978-3-319-74183-3>

Library of Congress Control Number: 2018930988

LNCS Sublibrary: SL2 – Programming and Software Engineering

© Springer International Publishing AG 2017

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

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

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

Cover illustration: Correspondences between the feedback loop and the MAPE-K Loop. Created by Marin Litou et al. Used with permission.

Printed on acid-free paper

This Springer imprint is published by Springer Nature

The registered company is Springer International Publishing AG

The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

An important concern for modern software systems is to become more cost-effective, while being versatile, flexible, resilient, dependable, energy-efficient, customizable, configurable, and self-optimizing when reacting to run-time changes that may occur within the system itself, its environment, or its requirements. One of the most promising approaches to achieving such properties is to equip software systems with self-managing capabilities using self-adaptation mechanisms. Self-adaptive systems are able to adjust their behavior or structure at run-time in response to their perception of the environment and the system itself. Lately, the software engineering community has recognized its key role in enabling the development of self-adaptive systems that are able to adapt to internal faults, changing requirements, and evolving environments. Despite recent advances in this area, one key aspect that remains to be tackled in depth is assurances, i.e., the provision of evidence that the system satisfies its stated functional and non-functional requirements during its operation in the presence of self-adaptation. The provision of assurances for self-adaptive systems is challenging since run-time changes introduce a high degree of uncertainty during their operation.

This book is one of the outcomes of Dagstuhl Seminar 13511 on “Software Engineering for Self-Adaptive Systems: Assurances” held in December 2013. It is the third book of the series on *Software Engineering for Self-Adaptive Systems*, and the papers in this volume address the state of the art in the field of assurances for self-adaptive systems, describe a wide range of approaches coming from different strands of software engineering and control engineering, and posit future challenges facing this field of research. More specifically, this book comprises a research challenges paper, three topical papers dealing in depth with the identified challenges, and invited papers from recognized experts in the field. All the papers in this book have been peer-reviewed, with the exception of the research challenges paper, which was written in several iterations over the past two years by the participants of this Dagstuhl Seminar. The book consists of four parts: “Research Challenges,” “Evaluation,” “Integration and Coordination,” and “Reference Architectures and Platforms.”

Part 1 of the book, entitled “Research Challenges,” contains four papers including the paper “Software Engineering for Self-Adaptive Systems: Research Challenges on Assurances,” and three working group papers that elaborate on the key issues raised on the challenges paper.

The challenges paper summarizes the Dagstuhl Seminar discussions and provides insights on the provision of assurances for self-adaptive software systems. Instead of dealing with a wide range of topics associated with the field, this paper focuses on three fundamental topics of self-adaptation regarding the provision of assurances: perpetual assurances, decomposing assurances, and lessons on assurances from control theory.

The second paper in the first part of this book by Weyns, Bencomo, Calinescu, Câmara, Ghezzi, Grassi, Grunske, Inverardi, Jezequel, Malek, Mirandola, Mori, and Tamburrelli, entitled “Perpetual Assurances in Self-Adaptive Systems,” provides a

background framework and the foundation for perpetual assurances for self-adaptive systems. The paper also discusses concrete challenges of offering perpetual assurances, requirements for solutions, realization techniques, and mechanisms to make solutions suitable.

The third paper entitled “Challenges in Decomposing and Composing Assurances for Self-Adaptive Systems,” authored by Schmerl, Andersson, Vogel, Cohen, Rubira, Brun, Gorla, Zambonelli, and Baresi, discusses how existing assurance techniques can be applied to composing and decomposing assurances for self-adaptive systems, highlights the challenges in applying them, summarizes existing research to address some of these challenges, and identifies gaps and opportunities to be addressed by future research.

The fourth paper “What Can Control Theory Teach Us About Assurances in Self-Adaptive Software Systems?,” authored by Litoiu, Shaw, Tamura, Villegas, Müller, Giese, Rouvoy, and Rutten, describes the control theory approach for the provision of assurances, explains several control strategies illustrated with examples from both domains (classical control theory and self-adaptive systems), and shows how the issues addressed by these strategies can and should be considered for the assurance of self-adaptive software systems.

Part 2 of this book, entitled “Evaluation,” consists of four papers describing verification and validation techniques that can be used as evidence for the provision of assurances.

The first paper by Sharifloo and Metzger, entitled “MCaaS: Model Checking in the Cloud for Assurances of Adaptive Systems,” introduces a cloud-based framework that delivers model checking as a service (MCaaS). MCaaS offloads computationally intensive model checking tasks to the cloud, thereby offering verification capabilities on demand. Self-adaptive systems running on any kind of connected device may take advantage of model checking at run-time by invoking the MCaaS service. As a proof of concept, the authors have implemented and validated their proposed approach for the case of probabilistic model checking.

The second paper, entitled “Analyzing Self-adaptation via Model Checking of Stochastic Games,” by Cámará, Garlan, Moreno and Schmerl describes an approach based on model checking of stochastic multiplayer games that enables developers to approximate the behavioral envelope of a self-adaptive system by analyzing best- and worst-case scenarios of alternative designs for self-adaptation mechanisms. Compared with other sources of evidence, such as simulations or prototypes, the proposed approach is purely declarative and hence has the potential of providing developers with a preliminary understanding of adaptation behaviour with less effort, and without the need to have any specific adaptation algorithms or infrastructure in place.

The third paper by Eberhardinger, Anders, Seebach, Siefert, Knapp and Reif, entitled “An Approach for Isolated Testing of Self-organization Algorithms,” describes a systematic approach for testing self-organization algorithms. A key feature of the proposed algorithm testing framework is automation since it is rarely possible to cope with the ramified state space manually. The test automation adopts a model-based testing approach, where probabilistic environment profiles are used to derive test cases that are performed and evaluated on isolated self-organization algorithms.

The last paper of this part by Calinescu, Gerasimou, Johnson and Paterson, entitled “Runtime Quantitative Verification — Advances, Applications and Research Challenges,” surveys recent advances in the development of efficient run-time quantitative verification techniques, the application of these techniques within multiple domains, and some outstanding research challenges.

Part 3 of the book covers “Integration and Coordination,” and includes three papers on approaches for coordinating interactions in self-adaptive software systems.

The first paper in this part is by Kříkava, Collet, Rouvoy, and Seinturier, and entitled “Contracts-Based Control Integration into Software Systems.” The described approach relies on the principles of design-by-contract to ensure the correctness and robustness of a self-adaptive software system built using feedback control loops. The proposed solution raises the level of abstraction upon which the loops are specified by allowing one to define and automatically verify system-level properties organized in contracts. These contracts are complemented by support for systematic fault handling.

The second paper, entitled “Synthesis of Distributed and Adaptable Coordinators to Enable Goal-Driven Choreography Evolution,” by Autili, Inverardi, Perucci, and Tivoli, proposes a method for the automatic synthesis of evolving choreographies in which the coordination software is synthesized in order to enable proxies and control in the interaction between participant services. The ability to evolve the coordination logic in a modular way enables choreography evolution in response to possible changes. A running example in the domain of intelligent transportation systems illustrates the proposed method.

The last paper in this section, entitled “Models for the Consistent Interaction of Adaptations in Self-Adaptive Systems,” by Cardozo, Mens, and Clarke, describes existing approaches that allow for the development of self-adaptive systems and management of the behavioral inconsistencies that may appear due to the interaction of adaptations at run-time. Each of these approaches is evaluated with respect to the assurances they provide for the run-time consistency of the system, in the light of dynamic behavior adaptations.

Part 4 of the book contains four papers covering a wide range of issues related to “Reference Architectures and Platforms.”

The first paper in this part is by Rutten, Marchand, and Simon, and is entitled “Feedback Control as MAPE-K Loop in Autonomic Computing.” This paper surveys the MAPE-K loop from the point of view of control theory techniques, and then discusses continuous and discrete (supervisory) control techniques, and their application, to the feedback control of computing systems. It also proposes detailed interpretations of feedback control loops as MAPE-K loops, and illustrates these interpretations using a variety of case studies.

The second paper, entitled “An Extended Description of MORPH: A Reference Architecture for Configuration and Behavior Self-Adaptation,” and authored by Braberman, D’Ippolito, Kramer, Sykes, and Uchitel provides an extended description of a reference architecture that allows for coordinated, yet transparent and independent, adaptation of system configuration and behavior, thus accommodating complex self-adaptation scenarios.

The final paper of this part, entitled “MOSES: A Platform for Experimenting with QoS-driven Self-Adaptation Policies for Service-Oriented Systems,” by Cardellini,

Casalicchio, Grassi, Iannucci, Lo Presti, and Mirandola describes a software platform supporting QoS-driven adaptation of service-oriented systems. The platform integrates within a unified framework different adaptation mechanisms, enabling a greater flexibility in facing various operating environments, and the possibly conflicting QoS requirements of several concurrent users.

We would like to thank all the authors of the book chapters for their contributions, the participants of the Dagstuhl Seminar 13511 on “Software Engineering for Self-Adaptive Systems: Assurances” for their inspiring participation in moving this field forward, and Alfred Hofmann and his team at Springer for believing in this important project and helping us to publish this book. Last but not least, we deeply appreciate the great efforts of the following expert reviewers who helped us ensure that the contributions are of high quality: J. Andersson, M. Autili, N. Bencomo, R. Calinescu, J. Camara, N. Cardozo, S. Clarke, P. Collet, V. Cortelessa, N. D’Ippolito, C. E. da Silva, A. Filieri, A. Gorla, V. Grassi, K. Johnson, N. Khakpour, F. Krikava, A. Leva, M. Litoiu, S. Mocanu, K. Mens, R. Mirandola, H. Muller, J. Mylopoulos, R. Rouvoy, E. Rutten, B. Schmerl, V. Souza, M. Tivoli, S. Uchitel, N. Villegas, D. Weyns, and several anonymous reviewers.

We hope that this book will prove valuable for both practitioners and researchers involved in the development and deployment of self-adaptive software systems.

January 2017

Rogério de Lemos
David Garlan
Carlo Ghezzi
Holger Giese

Contents

Research Challenges

Software Engineering for Self-Adaptive Systems: Research Challenges in the Provision of Assurances	3
<i>Rogério de Lemos, David Garlan, Carlo Ghezzi, Holger Giese, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Danny Weyns, Luciano Baresi, Nelly Bencomo, Yuriy Brun, Javier Camara, Radu Calinescu, Myra B. Cohen, Alessandra Gorla, Vincenzo Grassi, Lars Grunske, Paola Inverardi, Jean-Marc Jezequel, Sam Malek, Raffaela Mirandola, Marco Mori, Hausi A. Müller, Romain Rouvoy, Cecilia M. F. Rubira, Eric Rutten, Mary Shaw, Giordano Tamburrelli, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, and Franco Zambonelli</i>	
Perpetual Assurances for Self-Adaptive Systems	31
<i>Danny Weyns, Nelly Bencomo, Radu Calinescu, Javier Camara, Carlo Ghezzi, Vincenzo Grassi, Lars Grunske, Paola Inverardi, Jean-Marc Jezequel, Sam Malek, Raffaela Mirandola, Marco Mori, and Giordano Tamburrelli</i>	
Challenges in Composing and Decomposing Assurances for Self-Adaptive Systems	64
<i>Bradley Schmerl, Jesper Andersson, Thomas Vogel, Myra B. Cohen, Cecilia M. F. Rubira, Yuriy Brun, Alessandra Gorla, Franco Zambonelli, and Luciano Baresi</i>	
What Can Control Theory Teach Us About Assurances in Self-Adaptive Software Systems?	90
<i>Marin Litoiu, Mary Shaw, Gabriel Tamura, Norha M. Villegas, Hausi A. Müller, Holger Giese, Romain Rouvoy, and Eric Rutten</i>	

Evaluation

MCaaS: Model Checking in the Cloud for Assurances of Adaptive Systems	137
<i>Amir Molzam Sharifloo and Andreas Metzger</i>	
Analyzing Self-Adaptation Via Model Checking of Stochastic Games	154
<i>Javier Cámara, David Garlan, Gabriel A. Moreno, and Bradley Schmerl</i>	

An Approach for Isolated Testing of Self-Organization Algorithms	188
<i>Benedikt Eberhardinger, Gerrit Anders, Hella Seebach, Florian Siefert, Alexander Knapp, and Wolfgang Reif</i>	
Using Runtime Quantitative Verification to Provide Assurance Evidence for Self-Adaptive Software: Advances, Applications and Research Challenges	223
<i>Radu Calinescu, Simos Gerasimou, Kenneth Johnson, and Colin Paterson</i>	
Integration and Coordination	
Contracts-Based Control Integration into Software Systems	251
<i>Filip Kříkava, Philippe Collet, Romain Rouvoy, and Lionel Seinturier</i>	
Synthesis of Distributed and Adaptable Coordinators to Enable Choreography Evolution	282
<i>Marco Autili, Paola Inverardi, Alexander Perucci, and Massimo Tivoli</i>	
Models for the Consistent Interaction of Adaptations in Self-Adaptive Systems	307
<i>Nicolás Cardozo, Kim Mens, and Siobhán Clarke</i>	
Feedback Control as MAPE-K Loop in Autonomic Computing.	349
<i>Eric Rutten, Nicolas Marchand, and Daniel Simon</i>	
Reference Architectures and Platforms	
An Extended Description of MORPH: A Reference Architecture for Configuration and Behaviour Self-Adaptation	377
<i>Victor Braberman, Nicolas D'Ippolito, Jeff Kramer, Daniel Sykes, and Sebastian Uchitel</i>	
MOSES: A Platform for Experimenting with QoS-Driven Self-Adaptation Policies for Service Oriented Systems	409
<i>Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Stefano Iannucci, Francesco Lo Presti, and Raffaela Mirandola</i>	
Author Index	435

Research Challenges

Software Engineering for Self-Adaptive Systems: Research Challenges in the Provision of Assurances

Rogério de Lemos^{1(✉)}, David Garlan², Carlo Ghezzi³, Holger Giese⁴
(Dagstuhl Seminar Organizers),
Jesper Andersson, Marin Litoiu, Bradley Schmerl, Danny Weyns
(Section Coordinators),
Luciano Baresi, Nelly Bencomo, Yuriy Brun, Javier Camara, Radu Calinescu,
Myra B. Cohen, Alessandra Gorla, Vincenzo Grassi, Lars Grunske,
Paola Inverardi, Jean-Marc Jezequel, Sam Malek, Raffaela Mirandola,
Marco Mori, Hausi A. Müller, Romain Rouvoy, Cecília M. F. Rubira,
Eric Rutten, Mary Shaw, Giordano Tamburrelli, Gabriel Tamura,
Norha M. Villegas, Thomas Vogel, and Franco Zambonelli
(Dagstuhl Seminar Participants)

¹ University of Kent, Canterbury, UK
r.delemos@kent.ac.uk

² Carnegie Mellon University, Pittsburgh, USA

³ Politecnico di Milano, Milano, Italy

⁴ Hasso Plattner Institute for Software Systems Engineering, Potsdam, Germany

Abstract. The important concern for modern software systems is to become more cost-effective, while being versatile, flexible, resilient, dependable, energy-efficient, customisable, configurable and self-optimising when reacting to run-time changes that may occur within the system itself, its environment or requirements. One of the most promising approaches to achieving such properties is to equip software systems with self-managing capabilities using self-adaptation mechanisms. Despite recent advances in this area, one key aspect of self-adaptive systems that remains to be tackled in depth is the provision of assurances, *i.e.*, the collection, analysis and synthesis of evidence that the system satisfies its stated functional and non-functional requirements during its operation in the presence of self-adaptation. The provision of assurances for self-adaptive systems is challenging since run-time changes introduce a high degree of uncertainty. This paper on research challenges complements previous roadmap papers on software engineering for self-adaptive systems covering a different set of topics, which are related to assurances, namely, perpetual assurances, composition and decomposition of assurances, and assurances obtained from control theory. This research challenges paper is one of the many results of the Dagstuhl Seminar 13511 on *Software Engineering for Self-Adaptive Systems: Assurances* which took place in December 2013.

1 Introduction

Repairing faults, or performing upgrades on different kinds of software systems have been tasks traditionally performed as a maintenance activity conducted off-line. However, as software systems become central to support everyday activities and face increasing dependability requirements, even as they have increased levels of complexity and uncertainty in their operational environments, there is a critical need to improve their resilience, optimize their performance, and at the same time, reduce their development and operational costs. This situation has led to the development of systems able to reconfigure their structure and modify their behaviour at run-time in order to improve their operation, recover from failures, and adapt to changes with little or no human intervention. These kinds of systems typically operate using an explicit representation of their own structure, behaviour and goals, and appear in the literature under different designations (*e.g.*, self-adaptive, self-healing, self-managed, self-*, autonomic). In particular, self-adaptive systems should be able to modify their behavior and/or structure in response to their perception of the environment and the system itself, and their goals.

Self-adaptive systems have been studied independently within different research areas of software engineering, including requirements engineering, modelling, architecture and middleware, event-based, component-based and knowledge-based systems, testing, verification and validation, as well as software maintenance and evolution [14, 30]. On the one hand, in spite of recent and important advances in the area, one key aspect of self-adaptive systems that poses important challenges yet to be tackled in depth is the provision of assurances, that is, the collection, analysis and synthesis of evidence for building arguments that demonstrate that the system satisfies its functional and non-functional requirements during operation. On the other hand, the topic of assurances for software-based systems has been widely investigated by the dependability community, in particular when considered in the context of safety-critical systems. For these types of systems there is the need to build coherent arguments showing that the system is able to comply with strict functional and non-functional requirements, which are often dictated by safety standards and general safety guidelines [7]. However, distinct from conventional systems in which the assurances are provided in tandem with development, the provision of assurances for self-adaptive systems should also consider their operation because run-time changes (*e.g.*, resource variability) introduce a high degree of uncertainty.

In self-adaptive systems, since changes and uncertainty may affect the system during its operation, it is expected that assurances also need to be perpetually revised depending on the type and number of changes, and how the system self-adapts to these changes in the context of uncertainties. In order to allow the continuous revision of assurances, new arguments need to be formed based on new evidence or by composing or decomposing existing evidence. Concepts and abstractions for such evidence can be obtained from classical control theory for providing reasoning about whether the feedback loop enabling self-adaption is able to achieve desired properties. Moreover, feedback loops supported by

processes should provide the basis for managing the continuous collection, analysis and synthesis of evidence that will form the core of the arguments that substantiate the provision of assurances.

The goal of this paper on the provision of assurances for self-adaptive systems is to summarize the topic state-of-the-art and identify research challenges yet to be addressed. This paper does not aim to supersede the previous roadmap papers on software engineering of self-adaptive systems [14, 30], but rather to complement previous papers by focusing on assurances. Though assurances were lightly treated in the former papers, this paper goes in more depth on topics that pose concrete challenges for the development, deployment, operation, evolution and decommission of self-adaptive systems, and identifies potential research directions for some of these challenges.

In order to provide a context for this paper, in the following, we summarize the most important research challenges identified in the two previous roadmap papers. In the first paper [14], the four topics covered were: *modeling dimensions*, where the challenge was to define models that can represent a wide range of system properties; *requirements*, where the challenge was to define languages capable of capturing uncertainty at the abstract level of requirements; *engineering*, where the challenge was to make the role of feedback control loops more explicit; *assurances*, where the challenge was how to supplement traditional V&V methods applied at requirements and design stages of development with run-time assurances. In the second paper [30], the four topics covered were: *design space*, where the challenge was to define what is the design space for self-adaptive systems, including the decisions the developer should address; *processes*, where the challenge was to define innovative generic processes for the development, deployment, operation, and evolution of self-adaptive systems; *decentralization of control loops*, where the challenge was to define architectural patterns for feedback control loops that would capture a varying degree of centralization and decentralization of the loop elements; *practical run-time verification and validation (V&V)*, where the challenge was to investigate V&V methods and techniques for obtaining inferential and incremental assessments for the provision of assurances.

For the motivation and presentation of the research challenges associated with the provision of assurances when engineering self-adaptive systems, we divide this paper into three parts, each related to one of the identified key research challenges. For each key research challenge, we present a description of the topic related to the challenge, and suggest future directions of research around the identified challenges to the topic. The three key research challenges are: perpetual assurances (Sect. 2), composition and decomposition of assurances (Sect. 3), and what can we learn from control theory regarding the provision of assurances (Sect. 4). Finally, Sect. 5 summarizes our findings.

2 Perpetual Assurances

Changes in self-adaptive systems, such as changes of parameters, components, and architecture, are shifted from development time to run-time. Furthermore,

the responsibility for these activities is shifted from software engineers or system administrators to the system itself. Hence, an important aspect of the software engineering process for self-adaptive systems, in particular business or safety critical systems, is providing new evidence that the system goals are satisfied during its entire lifetime, from inception to and throughout operation until decommission. The state of the art advocates the use of formal models as one promising approach to providing evidence for goal compliance. Several approaches employ formal methods to provide such guarantees by construction. In particular, recent research suggests the use of probabilistic models to verify system properties and support decision-making about adaptation at run-time. However, providing assurances for the goals of self-adaptive systems that must be achieved during the entire lifecycle remains a difficult challenge. This section summarizes a background framework for providing assurances for self-adaptive systems that we term “perpetual assurances for self-adaptive systems.” We discuss uncertainty as a key challenge for perpetual assurances, requirements for solutions addressing this challenge, realization techniques and mechanisms to make these solutions effective, and benchmark criteria to compare solutions. For an extensive description of the background framework for perpetual assurance, we refer the reader to [46].

2.1 Uncertainty as a Key Challenge for Perpetual Assurances

We use the following working definition for *perpetual assurances for self-adaptive systems*:

Perpetual assurances for self-adaptive systems mean providing evidence for requirements compliance through an enduring process that continuously provides new evidence by combining system-driven and human-driven activities to deal with the uncertainties that the system faces across its lifetime, from inception to operation in the real world.

Thus, providing assurances cannot be achieved by simply using off-line solutions, possibly complemented with on-line solutions. Instead, we envision that perpetual assurances will employ a continuous process where humans and the system jointly and continuously derive and integrate new evidence and arguments required to assure stakeholders (*e.g.*, end users and system administrators) that the requirements are met by the self-adaptive system despite the uncertainties it faces throughout its lifetime.

A primary underlying challenge for this process stems from *uncertainty*. There is no agreement on a definition of uncertainty in the literature. Here we provide a classification of sources of uncertainty based on [33] using two dimensions of a taxonomy for uncertainty proposed in [35], *i.e.*, *location* and *nature*. *Location* refers to where uncertainty manifests in the description (the model) of the studied system or phenomenon. We can specialize it into: (i) input parameters, (ii) model structure, and (iii) context. *Nature* indicates whether the uncertainty is due to the lack of accurate information (*epistemic*) or to the

inherent variability of the phenomena being described (*aleatory*). Recognizing the presence of uncertainty and managing it can mitigate its potentially negative effects and increase the level of assurance in a self-adaptive system. By ignoring uncertainties, one could draw unsupported claims on system validity or generalize them beyond their bounded scope.

Table 1 shows our classification of uncertainty based on [33], which is inspired by recent research on self-adaptation, *e.g.*, [18, 21]. Each source of uncertainty is classified according to the location and nature dimensions of the taxonomy.

Table 1. Classification of sources of uncertainty based on [33].

Source of Uncertainty		Classification	
		Location	Nature
Simplifying assumptions	System	Structural/context	Epistemic
Model drift		Structural	Epistemic
Incompleteness		Structural	Epistemic/Aleatory
Future parameters value		Input	Epistemic
Adaptation functions		Structural	Epistemic/Aleatory
Automatic learning		Structural/input	Epistemic/Aleatory
Decentralization		Context/structural	Epistemic
Requirements elicitation	Goals	Structural/input	Epistemic/Aleatory
Specification of goals		Structural/input	Epistemic/Aleatory
Future goal changes		Structural/input	Epistemic/Aleatory
Execution context	Context	Context/structural/input	Epistemic
Noise in sensing		Input	Epistemic/Aleatory
Different sources of information		Input	Epistemic/Aleatory
Human in the loop	Human	Context	Epistemic/Aleatory
Multiple ownership		Context	Epistemic/Aleatory

Sources of uncertainty are structured in four groups: (i) uncertainty related to the *system* itself; (ii) uncertainty related to the system *goals*; (iii) uncertainty in the execution *context*; and (iv) uncertainty related to *human* aspects. Uncertainty in its various forms represents the ultimate source of both motivations for and challenges to perpetual assurance. Uncertainty manifests through *changes*. For example, uncertainty in capturing the precise behavior of an input phenomenon to be controlled results in assumptions made during the implementation of the system. Therefore, the system must be calibrated later when observations of the physical phenomenon are made. This in turn leads to changes in the implemented control system that must be scrutinized to derive assurances about its correct operation.

2.2 Requirements for Solutions that Realize Perpetual Assurances

The provision of perpetual assurance for self-adaptive systems must cope with a variety of uncertainty sources that depend on the purpose of self-adaptation

and the environment in which the assured self-adaptive system operates. To this end, they must build and continually update their assurance arguments through the integration of two types of assurance evidence. The first type of evidence corresponds to system and environment components not affected significantly by uncertainty, and therefore can be obtained using traditional off-line approaches [28]. The second type of evidence is associated with the system and environment components affected by the sources of uncertainty summarized in Table 1. This type of evidence is required for each of the different functions of adaptation: from sensing to monitoring, analyzing, planning, executing, and activating [15]. The evidence must be synthesized at run-time, when the uncertainty is treated, *i.e.*, reduced, quantified or resolved sufficiently to enable such synthesis.

Table 2 summarizes the requirements for perpetual assurance solutions. R1–R7 are functional requirements to treat uncertainty. R8–R10 are non-functional requirements to provide assurance that are timely, non-intrusive and auditable.

2.3 Approaches to Perpetual Assurances

Several approaches have been developed in previous decades to check whether a software system complies with its requirements. Table 3 below gives an overview of these approaches organized in three categories: human-driven approaches (manual), system-driven (automated), and hybrid (manual and automated).

We briefly discuss one representative approach of each group. ***Formal proof*** is a human-driven approach that uses a mathematical calculation to prove a sequence of related theorems that refer, or are based upon, a formal specification of the system. Formal proofs are rigorous and unambiguous, but can only be produced by experts with both detailed knowledge about how the self-adaptive system works and significant mathematical experience. As an example, in [51] the authors formally prove a set of theorems to assure safety and liveness properties of self-adaptive systems. The approach is illustrated for data stream components that modify their behavior in response to external conditions through the insertion and removal of filters. ***Run-time verification*** is a system-driven approach that is based on extracting information from a running system to detect whether certain properties are violated. Run-time verification is less complex than traditional formal verification because only one or a few execution traces are analyzed at a time. As an example, in [40] the authors introduce an approach for estimating the probability that a temporal property is satisfied by a run of a program. ***Model checking*** is a well-known hybrid approach that allows designers to check that a property holds for all reachable states in a system. Model checking can be applied off-line or on-line, and can only work in practice on a high-level abstraction of an adaptive system or on one of its components. For example, [25] models MAPE loops of a mobile learning application as timed automata and verifies robustness requirements expressed in timed computation tree logic using the UPPAAL tool. In [10] QoS requirements of service-based systems are expressed as probabilistic temporal logic formulae, which are automatically analyzed at run-time to identify optimal system configurations.

Table 2. Summary requirements.

Requirement	Brief description
R1: Monitor uncertainty	A perpetual assurance solution must continually observe the sources of uncertainty affecting the self-adaptive system
R2: Quantify uncertainty	A perpetual assurance solution must use its observations of uncertainty sources to continually quantify and potentially mitigate the uncertainties affecting its ability to provide assurance evidence
R3: Manage overlapping uncertainty sources	A perpetual assurance solution must continually deal with overlapping sources of uncertainty and may need to treat these sources in a composable fashion
R4: Derive new evidence	A perpetual assurance solution must continually derive new assurance evidence arguments
R5: Integrate new evidence	A perpetual assurance solution must continually integrate new evidence into the assurance arguments for the safe behavior of the assured self-managing system
R6: Combine new evidence	A perpetual assurance solution may need to continually combine new assurance evidence synthesized automatically and provided by human experts
R7: Provide evidence for the components and activities that realize R1-R6	A perpetual assurance solution must provide assurance evidence for the system components, the human activities, and the processes used to meet the previous set of requirements
R8: Produce timely updates	The activities carried out by a perpetual assurance solution must produce timely updates of the assurance arguments
R9: Limited overhead	The activities of the perpetual assurance solution and their overheads must not compromise the operation of the assured self-adaptive system
R10: Auditable arguments	The assurance evidence produced by a solution and the associated assurance arguments must be auditable by human stakeholders

Table 3. Approaches for assurances.

Assurances approach category	Examples
Human-driven approaches	Formal proof Simulation
System-driven approaches	Run-time verification Sanity checks Contracts
Hybrid approaches	Model checking Testing

2.4 Mechanisms for Turning Perpetual Assurances into Reality

Turning the approaches of perpetual assurances into a working reality requires aligning them with the requirements discussed in Sect. 2.2. For the functional requirements (R1 to R7), the central problem is how to build assurance arguments based on the evidence collected at run-time, which should be composed with the evidence acquired throughout the lifetime of the system, possibly by different approaches. For the quality requirements (R8 to R10) the central problem is to make the solutions efficient and to support the interchange of evidence between the system and its users. Efficiency needs to take into account the size of the self-adaptive system and the dynamism it is subjected to. An approach for perpetual assurances is efficient if it is able to: (i) provide results (assurances) within defined time constraints (depending on the context of use); (ii) consume an acceptable amount of resources, so that the resources consumed over time (*e.g.*, memory size, CPU, network bandwidth, energy, etc.) remain within a limited fraction of the overall resources used by the system; and (iii) scale well with respect to potential increases in size of the system and the dynamism it is exposed to.

It is important to note that orthogonal to the requirements for perceptual assurance in general, the level of assurance that is needed depends on the requirements of the self-adaptive system under consideration. In some cases, combining regular testing with simple and time-effective run-time techniques, such as sanity checks and contract checking, will be sufficient. In other cases, more powerful approaches are required. For example, model checking could be used to verify a safe envelope of possible trajectories of an adaptive system at design time, and verification at run-time to check whether the next change of state of the system keeps it inside the pre-validated envelope. We briefly discuss two classes of mechanisms that can be used to provide the required functionalities for perpetual assurances and meet the required qualities: decomposition mechanisms and model-driven mechanisms.

Decomposition Mechanisms. The first class of promising mechanisms for the perpetual provisioning of assurances is based on the principle of decomposition, which can be carried out along two dimensions:

1. Time decomposition, in which: (i) some preliminary/intermediate work is performed off-line, and the actual assurance is provided on-line, building on these intermediate results; (ii) assurances are provided with some degree of approximation/coarseness, and can be refined if necessary.
2. Space decomposition, where verification overhead is reduced by independently verifying each individual component of a large system, and then deriving global system properties through verifying a composition of its component-level properties. Possible approaches that can be used are: (i) flat approaches, that exploit only the system decomposition into components; (ii) hierarchical approaches, where the hierarchical structure of the system is exploited; (iii) incremental approaches targeted at frequently changing systems, in which re-verification are carried out only on the minimal subset of components affected by a change.

Model-based Mechanisms. For any division of responsibilities between human and systems in the perpetual assurance process, an important issue is how to define in a traceable way the interplay between the actors involved in the process. Model-driven mechanisms support the rigorous development of a self-adaptive system from its high-level design up to its running implementation, and they support traceable modifications of the system by humans and/or of its self-adaptive logic, *e.g.*, to respond to modifications of the requirements. In this direction, [45] presents a domain-specific language for the modeling of adaptation engines and a corresponding run-time interpreter that drives the adaptation engine operations. The approach supports the combination of on-line machine-controlled adaptations and off-line long-term adaptations performed by humans to maintain and evolve the system. Similarly, [24] proposes an approach called ActivFORMS in which a formal model of the adaptation engine specified in timed automata and adaptation goals expressed in timed computation tree logic are complemented by a virtual machine that executes the verified models, guaranteeing at run-time compliance with the properties verified off-line. The approach supports on-the-fly deployment of new models by human experts to deal with new goals.

2.5 Benchmark Criteria for Perpetual Assurances

We provide benchmark criteria for comparing four key aspects of perpetual assurance approaches: approach capabilities, basis of evidence for assurances, stringency of assurances, and performance. The criteria, shown in Table 4, cover both functional and quality requirements for perpetual assurances approaches.

Several criteria from Table 4 directly map to a requirement from Table 2. For example, ‘Timeliness’ directly links to requirement R8 (‘Produce timely updates’). Other criteria correspond to multiple requirements. For example, ‘Human evidence’ links to R5 (‘Integrate new evidence’), R7 (‘Provide evidence for human activities that realize R5’), and R10 (‘Auditable arguments’). Other arguments only link indirectly to requirements from Table 2. This is the case for the ‘Handling alternatives’ criterion, which corresponds to the solution for self-adaptation, which may provide different levels of support for the requirements of perpetual assurances.

2.6 Research Challenges

Assuring requirements compliance of self-adaptive systems calls for an enduring process where evidence is collected over the lifetime of the system. This process for the provision of perpetual assurances for self-adaptive systems poses four key challenges.

First, we need a better understanding of the nature of uncertainty for software systems and of how this translates into requirements for providing perpetual assurances. Additional research is required to test the validity and coverage of this set of requirements.

Table 4. Summary of benchmark aspects and criteria for perpetual assurances.

Benchmark Aspect	Benchmark Criteria	
	Criteria	Description
Capabilities of approaches to provide assurances	Variability	Capability of an approach to handle variations in requirements (adding, updating, deleting goals), and the system (adding, updating, deleting components)
	Inaccuracy & incompleteness	Capability of an approach to handle inaccuracy and incompleteness of models of the system and context
	Competing criteria	Capability of an approach to balance the tradeoffs between utility (<i>e.g.</i> , coverage, quality) and cost (<i>e.g.</i> , time, resources)
	User interaction	Capability of an approach to handle changes in user behavior (preferences, profile)
	Handling alternatives	Capability of an approach to handle changes in adaptation strategies (<i>e.g.</i> , pre-emption)
Basis of assurance benchmarking	Historical data only	Capability of an approach to provide evidence over time based on historical data
	Projections in the future	Capability of an approach to provide evidence based on predictive models
	Combined approaches	Capability of an approach to provide evidence based on combining historical data with predictive models
	Human evidence	Capability of an approach to complement automatically gathered evidence by evidence provided by humans
Stringency of assurances	Assurance rational	Capability of the approach to provide the required rational of evidence for the purpose of the system and its users (<i>e.g.</i> , completeness, precision)
Performance of approaches	Timeliness	The time an approach requires to achieve the required evidence
	Computational overhead	The resources required by an approach (<i>e.g.</i> , memory and CPU) for enacting the assurance approach
	Complexity	The scope of applicability of an approach to different types of problems

Second, we need a deeper understanding of how to monitor and quantify uncertainty. In particular, how to handle uncertainty in the system, its goal and its environment remains to a large extent an open research problem.

Third, the derivation and integration of new evidence pose additional hard challenges. Decomposition and model-based reasoning mechanisms represent potential approaches for moving forward. However, making these mechanisms effective is particularly challenging and requires a radical revision of many existing techniques.

Last but not least, to advance research on assurances for self-adaptive systems, we need self-adaptive system exemplars (*e.g.* [28, 47]) that can be used to assess the effectiveness of different solutions.

3 Composing and Decomposing Assurances

Assuring a self-adaptive system in all the configurations that it could possibly be in, under all the environments it can encounter, is challenging. One way to address this challenge is to understand how to *decompose assurances* so that an entire revalidation is not required at run-time when the system changes. Another source of challenges for assuring self-adaptive systems is when they are composed together to create larger systems (for example, having multiple adaptive systems in autonomous vehicles, or having multiple adaptive systems managing a building). Typically, assurances are also required for this systems-of-systems context. We therefore need ways to *compose assurances* that do not require complete revalidation of each of the constituent parts.

For safety-critical systems there is a large body of work on constructing safety cases [8], or more generally assurance cases [6], that allow engineers to build assurance arguments to provide confidence that a system will be safe (in addition to other qualities). How these assurance cases are constructed for safety-critical systems can shed some light on how to provide assurances for self-adaptive systems. Typically, building assurance cases involve decomposing top level goals into argumentation structures that involve sub-goals, strategies for achieving the goals, and defining evidence that can be collected to show that the goals are achieved. For example, a safety case presents a structured demonstration that a system is acceptably safe in a given context – *i.e.*, it is a comprehensive presentation of evidence linked by argument to a claim. Structuring evidence in such a way means that an expert can make a judgment that the argument makes sense and thus, if the evidence in the case is provided, have confidence that the system is acceptably safe. Assurance cases are a generalization of safety cases to construct arguments that are about more than just safety.

Assurance cases themselves can be composed together to provide assurances about a system with multiple goals, to reuse some assurances for goals in similar systems, or to provide assurances in systems-of-systems contexts. We can therefore use work on assurance case construction and composition as a guide to how to decompose and compose assurances for self-adaptive systems. For an extensive description of the ideas presented in this section, we refer the reader to [37].

3.1 Assurances in Self-adaptive Systems

While the focus of much of the research in self-adaptive systems to date has been to engineer systems that can maintain stated goals, especially in the presence of uncertain and changing environments, there is existing research in assurances for self-adaptive systems that either addresses how to compose assurances, or

can be used as part of an argument in assurance cases. We categorize existing work into the following areas:

Evidence types and sub-goals for use in assurance case decomposition.

Each of the classic activities of self-adaptation—monitoring, analysis, planning, and execution—have existing techniques that help to provide evidence for goals that can be used in assurance cases. For example, [11, 12] provide theoretical foundations based on information theory for determining if a self-adaptive system has enough information to diagnose problems. In [1], contextual goals can be used to identify modeling requirements. Models and simulations can provide evidence about whether adaptation should be done. Models (particularly performance models) have been used in [16, 31], for example. Formal models have also been used to provide evidence that adaptations will achieve desired results, for example probabilistic models in [19, 20, 38].

Assurance composition based on the MAPE-K loop. Once assurances have been decomposed, and evidence sources have been identified, we need ways to recompose the assurances. Fortunately, in self-adaptive systems, there is research that takes advantage of the common MAPE-K pattern used for constructing self-adaptive systems. The integration of V&V tasks into the MAPE-K loop is discussed in [41]. Both [41, 48] discuss the different modes of self-adaptive systems (called viability zones) that can guide what assurance techniques can be used in and between each mode.

Combining these approaches with work on assurance cases can lead to a principled way of designing, structuring, and adapting assurances for self-adaptive systems.

Decomposing Assurances in Self-adaptive Systems. Assurance cases naturally guide how to decompose assurances into subgoals and evidence. For self-adaptive systems there are a number of challenges centered on (a) what types

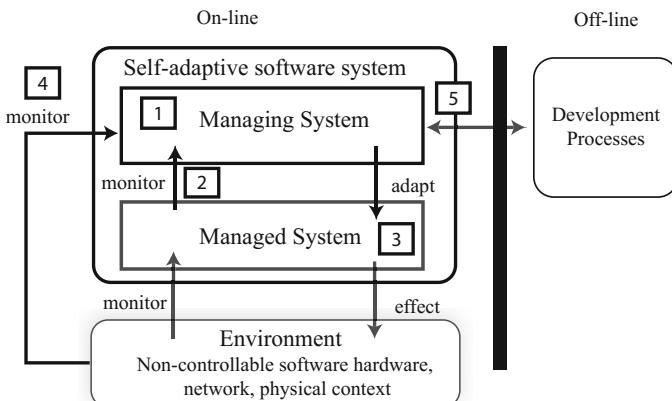


Fig. 1. Architectural reference model for self-adaptive software systems.

of evidence can be provided, and (b) where that evidence might come from. A reference model for self-adaptive systems, depicted in Fig. 1, can be used as a guide for addressing these challenges. This reference model can help to identify architectural concerns for self-adaptation that require assurances and that should be included in the argumentation structure. For example, we must provide convincing evidence that the managing system:

- makes a correct decision about when and how to adapt the managed system (cf. **1** in Fig. 1),
- correctly monitors the managed system **2** and the environment **4** and that the assurances and assumptions provided for the managed system and environment are correct such that the managing system can rely on them,
- correctly adapts the managed system **3**, which in turn must change according to this adaptation,
- correctly interacts with the development process **5** through which engineers directly adapt the managed system, or change the managing system itself (*e.g.*, to add new adaptation strategies).

This model can guide strategy selection and evidence placement for both functional and extra-functional goal decomposition. Within this, though, there are additional issues to consider when determining the argumentation structures:

Time and Lifecycle. Evidence for self-adaptive systems can be generated during development or at run-time. On-line techniques can be embedded in the managing systems to provide evidence at run-time. Off-line techniques are traditionally used during development or maintenance to provide static evidence. The managing system may, however, involve off-line techniques at run-time as an alternative for resource demanding evidence generation. The work presented in [36] discusses how one might use off-line formal verification results when reusing components; such work could also inspire formal verification reuse during run-time.

Independence and additivity. The inherent flexibility in a self-adaptive system suggests that argumentation structures be generated and adapted dynamically to reflect system adaptations. One aspect that promotes dynamic argumentation is independent evidence, which can be added to support arguments. We discuss this matter in more detail below. However, it is clear that if two or more evidence types are equally strong, the more independent should be favored. This will reduce the complexity of combining evidence for parts when assuring compositions.

Evidence completeness. Complete evidence is key for assurance cases. The argumentation structure should provide convincing arguments. For a self-adaptive system, completeness is affected by time and lifecycle concerns. As mentioned above, some evidence is not generated before run-time, thus complete evidence will be unavailable during development. In fact, completeness can change during the lifecycle. Evidence violations may, for instance, trigger adaptations, which implies that from time to time, evidence may transition from complete to incomplete and back to complete after a successful adaptation.

Composing Assurances in Self-adaptive Systems. The argumentation structures with goals, strategies, and evidence have a close analogy with validation & verification. For example, individual decomposed assurance cases have similarities with unit tests, while composing assurances is more closely aligned with integration and system testing. Unit tests are run without the global view of the system, and may either over- or under-approximate system behavior, while integration and system tests consider the environment under which they are run. Test dependencies, a subfield of software testing, provides some pointers to root causes for dependencies and consequences on a system’s testability and design for testability is mentioned as a viable resolution strategy. Dependencies between goals, strategies, and evidence have similar negative effects on the assurance case composition. We exemplify some causes and their effects in our analysis of three composition scenarios below.

1. Combining assurance cases for different goals of the same system: Consider the managing system in Fig. 1 and two goals, one for self-optimization and one for self-protection. The decomposition strategy described above generates two separate argumentation structures, one for each goal. We may not find the two top-level goals conflicting, however, parts in the argumentation structures may have explicit or implicit dependencies. For example, the inner components in the MAPE-K loop create numerous implicit and explicit inter-dependencies, which impact composability negatively.
2. Combining assurance cases for two different systems: In this case, we consider a situation where two independently developed systems are composed. We need to examine goal, evidence, and resource dependencies the composition creates. For example, some goals may subsume other goals, *i.e.*, the weakest claim needs to be replaced with the strongest claim. Further analysis of resource and evidence dependencies will indicate if evidence is independent and may be reused or if new evidence is required for the composition.
3. Combining multiple assurances for multiple systems composed in a systems-of-systems context: This is the extreme case of scenario 2. The analysis will be more complex and hence, also conflict resolution and evidence generation.

These issues are also challenging for the assurance community. Work in assurance case modularization [26, 50] can address both assurance case decomposition and composition through the use of contracts. Contracts and modularization of assurance cases will help with independence and additivity. Furthermore, [7] points out that assurance case decomposition is seldom explicit and that the assurance case community needs to develop rigorous decomposition strategies. These efforts should be tracked so that insights can be transferred to self-adaptive systems.

3.2 Research Challenges

In this section we have proposed that assurance cases can be used to guide decomposition and composition of assurances for self-adaptive system. We have

shown how self-adaptive systems themselves might help in informing how to decompose and compose assurance cases, and suggested that the assurance case community is addressing some of the challenges raised. However, there are a number of challenges that arise when trying to apply assurance cases to self-adaptation, which researchers in this area should investigate further.

Uncertainty. Self-adaptive systems are often self-adaptive because they are deployed in environments with uncertainty. This uncertainty affects the types of evidence that can be collected to support assurances, the ways in which the evidence can be collected, and even the specification of the assurance case itself. For example, goals in assurance cases need to specify the environmental assumptions under which they are valid, but for self-adaptive systems we need some way to make uncertainty about these assumptions first-class.

Assurance case dependencies. Goals, strategies, and evidence create a complex dependency web that connects argumentation structures. This web impacts how we derive and combine assurance cases negatively. A better understanding of the nature of these dependencies and how to mitigate their consequences will improve current practice for assurance case decomposition and composition. Existing work on testability and reuse of validation and verification results could be the point of departure.

Adaptation assurances. When conditions change and the system adapts, an assurance may describe how quickly or how well it adapts. For example, increased demand may trigger the addition of a web server. An assurance may state that when the per-server load exceeds a threshold, the system adapts within two minutes by adding web servers and the per-server load falls below the threshold within five minutes. This assurance may hold at all times, or may be expected to hold only when the demand increases but then remains constant.

Automatable assurance cases. Assurance cases rely on human judgment to discern whether the argument and rationale actually makes the case given the evidence. One of the aims of self-adaptation is to eliminate or at least reduce the involvement of humans in the management of a software system. To accomplish this, self-adaptation requires ways to computationally reason about assurance cases, and a logic to judge whether an assurance case is still valid, what changes must be made to it in terms of additional evidence, etc.

Adaptive assurances. As just alluded to, self-adaptation may require the assurance cases themselves to adapt. For example, replacing a new component in the system may require replacing evidence associated with that component in the assurance case. Changing goals of the system based on evolving business contexts will likely involve changes to the assurance cases for those goals. Automatable assurance cases are an initial step to addressing this challenge, but approaches, rules, and techniques for adapting the assurance cases themselves are also needed.

Assurance processes for self-adaptive software systems. One overarching challenge is the design of adequate assurance processes for self-adaptive systems. Such a process connects the system's goals, the architecture, and

implementation realizing the goals to the assurance cases' argumentation structures, its strategies, evidence types, and assurance techniques. This challenge requires that parts of the design and assurance process that were previously performed off-line during development time must move to run-time and be carried out on-line in the system itself. The assurance goals of a system are dependent on a correct, efficient, and robust assurance process, which employs on-line and off-line activities to maintain continuous assurance support throughout the system lifecycle. Currently, such processes are not sufficiently investigated and understood.

Reassurance. If we are able to move the evaluation of assurance cases to run-time, the challenge arises in how to reassure the system when things change. Reassurance may need to happen when environment states, or the state of the system itself, change. Which part of the assurance case needs to be re-evaluated? For composition, where the composition itself is dynamic, we need ways to identify the smallest set of claims (goals) that have to be reassured when two systems are composed? Which evidence needs to be re-established, and which can be reused?

4 Control Theory and Assurances

To realize perpetual assurances for adaptive systems requires effective run-time instrumentation to regulate the satisfaction of functional and non-functional requirements, in the presence of context changes and uncertainty (cf. Table 1). Control theory and feedback loops provide a number of powerful mechanisms for managing uncertainty in engineering adaptive systems [34]. Basically, feedback control allows us to manage uncertainty by monitoring the operation and environment of the system, comparing the observed variables against static or dynamic values to achieve (*i.e.*, system goals), and adjusting the system behavior to counteract disturbances that can affect the satisfaction of system requirements and goals. While continuous control theory suffices for purely physical systems, for cyber physical systems with significant software components a mix of discrete and continuous control is required. Moreover, adaptive systems require adaptive control where controllers must be modified at run-time. Many exciting and challenging research questions remain in applying control theory and concepts in the realm of self-adaptive systems.

The work presented in this paper is fundamentally based on the idea that, even for software systems that are too complex for direct application of classical control theory, the concepts and abstractions afforded by control theory can be useful. These concepts and abstractions provide design guidance to identify important control characteristics, as well as to determine not only the general steps but also the details of the strategy that determines the controllability of the resulting systems. This in turn enables careful reasoning about whether the control characteristics are in fact achieved in the resulting system.

Feedback loops have been adopted as cornerstones of software-intensive self-adaptive systems [9, 27, 30]. Building on this, this paper explores how classical

feedback loops, as defined by control theory, can contribute to the design of self-adaptive systems, particularly to their assurances. The proposed approach focuses on the abstract characteristics of classical feedback loops — including their formulation and afforded assurances, as well as the analysis required to obtain those assurances. The approach concentrates on the conceptual rather than the implementation level of the feedback-loop model. We investigated the relationships among desired properties that can be ensured by control theory applied in feedback loops (*e.g.*, stability, accuracy, settling time, or efficient resource use), the ways computational processes can be adjusted, the choice of the control strategy, and the quality attributes (*e.g.*, responsiveness, latency, or reliability) of the resulting system.

On the one hand, we discuss how feedback loops contribute to providing assurances about the behavior of the controlled system, and on the other hand, how the implementation of feedback loops in self-adaptive systems improves the realization of assurances in them. To set the stage for identifying concrete challenges, we first reviewed the major concepts of traditional control theory and engineering, then studied the parallels between control theory [3, 34] and the more recent research on feedback control of software systems (*i.e.*, MAPE-K loops and hierarchical arrangements of such loops) [22, 27] in the realm of self-adaptive systems. To gain a good understanding of the role that feedback loops play in the providing assurances for self-adaptive systems, the following books and seminal papers [2, 3, 9, 17, 27, 41, 43] are recommended.

In the next sections, we introduce basic concepts and properties that can be borrowed from control theory to provide assurances for self-adaptive systems. Then, we discuss research challenges and questions identified in analyzing classical feedback loops to guarantee desired properties for self-adaptive systems. For an extensive description of the ideas presented in this section, we refer the reader to [32].

4.1 Feedback Control

In a simple feedback control system, a process P (*i.e.*, the system to be adapted or managed in the self-adaptive systems realm) has a tuning parameter u (*e.g.*, a knob, which is represented by the little square box in Fig. 2) that can be manipulated by a controller C to change the behavior of the process, and a tracked metric y that can be sensed in some way. The system is expected to maintain the tracked metric y at a reference level y_r (*i.e.*, reference input), as illustrated in Fig. 2. Being a function that depends on time, C compares the value of y (subject to possible signal translation in the transducer) to the desired value y_r . The difference is the tracking or control error. If this error is significantly enough, the controller changes parameter u to drive the process in such a way as to reduce the tracking error. The tuning parameter u is often called the “control input” and the tracked metric y is often called the “measured output.”

Control theory depends on the definition of reference control points to specify system behavior and corresponding explicit mathematical models. These models describe the Process (P), the Controller (C) and the overall feedback system.

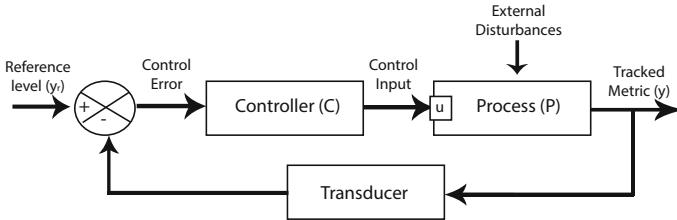


Fig. 2. The feedback loop control model.

Control theory separates between the design of the process to be controlled and the design of the controller. Given a model of a process, a controller is designed to achieve a particular goal (*e.g.*, stability, robustness). Control theory also provides assurances when the process and the whole system are described by models.

Feedback, as implemented in controlled systems as described above, is especially useful when processes are subject to unpredictable disturbances. In computing environments, disturbance examples include, among others, system loads, such as those implied by the number of users or request arrival rates, and variable hit rates on system caches. Feedback can also be useful when the computation in the process itself is unpredictable and its accuracy or performance can be tuned by adjusting its parameters.

4.2 Adaptive and Hierarchical Control

For systems that vary over time or face a wide range of external disturbances, it is impossible to design one controller that addresses all those changes. In these cases, there is the need to design a set of controllers (*i.e.*, parameterized controllers). When the current controller becomes ineffective, we switch to a new controller or adjust its parameters. When the controller is updated while the system runs, this control strategy is referred to as *adaptive control*. This control strategy requires additional logic that monitors the effectiveness of the controller under given conditions and, when some conditions are met, it re-tunes the controller to adapt it to the new situation. The control community has investigated adaptive control since 1961 according to Åström and Wittenmark [3]. They provide a working definition for adaptive control that has withstood the test of time: “An adaptive controller is a controller with adjustable parameters and a mechanism for adjusting the parameters.” This definition implies *hierarchical control*: arrangements of two (or more) layers of control loops, usually three-layered architectures.

Control theory offers several approaches for realizing adaptive control. Two of them are *model reference adaptive control* (MRAC) and *model identification adaptive control* (MIAC) [17]. MRAC and MIAC feature an additional controller that modifies the underlying controller that affects the target system. This higher-level controller, also referred to as the “adaptation algorithm”, is specified at design time for MRAC (*e.g.*, using simulation) and identified at

run-time (*e.g.*, using estimation techniques) for MIAC. In practice, given the dynamic nature of the reference model in MIAC, this approach is used more often for highly uncertain scenarios. The strategies for changing the underlying controller range from changing parameters (*e.g.*, three parameter gains in a PID controller) to replacing the entire software control algorithm (*e.g.*, from a set of predefined components).

In the seventies, the AI and robotics communities introduced three-layer intelligent hierarchical control systems (HICS) [39]. The *Autonomic Computing Reference Architecture* (ACRA), proposed by Kephart and Chess [23, 27] for the engineering of autonomic systems, is the most prominent reference architecture for hierarchical control. self-adaptive systems based on ACRA are defined as a set of hierarchically structured controllers. Using ACRA, software policies and assurances are injected from higher layers into lower layers. Other notable three-layer hierarchical control reference models include the Kramer and Magee model [29], DYNAMICO by Villegas *et al.* [44], and FORMS by Weyns *et al.* [49]. Please refer to Villegas *et al.* [42] for a more detailed discussion of these hierarchical control models for self-adaptive systems. The models at run-time (MART) community has developed extensive methods and techniques to evolve models dynamically for adaptive and hierarchical control purposes [5, 15].

4.3 Control Theory Properties

Exploiting control theory to realize effective assurances in self-adaptive systems implies that special attention needs to be paid to the selection of a “control strategy” that contributes to guaranteeing desired *properties*. From this perspective, the control strategy is even more critical than the mechanisms used to adjust the target system. Furthermore, a property that is properly achieved effectively becomes an assurance for the desired system behavior. We submit that the lessons learned from control theory in the assurance of desired properties is a promising research direction. Here, we present an overview of control theory properties (Villegas *et al.* comprehensively define these properties in the context of self-adaptive systems [43]). These properties, even if not captured in formalized models for self-adaptive systems, must be considered by their designers along the system’s engineering lifecycle.

Broadly, control theory studies two types of control loops: open and closed loops. Open loop models focus only on the controlled or managed system, that is, the outputs of the controlled system (*i.e.*, measured outputs) are not considered to compute the control input. In contrast, in closed loop models control inputs are computed from measured outputs.

Properties of the Open Loop Model. In the open loop model, the most important properties are stability, observability and controllability.

Stability means that for bounded inputs (commands or perturbations), the system will produce bounded state and output values. Unfortunately, perturbations are the enemy of stability in open loop self-adaptive systems, because

the system is not set up to recognize how much the perturbations affect the system. If the open self-adaptive systems is not stable, it can be stabilized through the design of a suitable controller. However, by analyzing the stability of the open system, we must understand the source of instability and design the controller appropriately.

Observability is the property of the model that allows to find, or at least estimate, the internal state variables of a system from the tracked output variables.

Controllability (or state controllability) describes the possibility of driving the open system to a desired state, that is, to bring its internal state variables to certain values [13]. While observability is not a necessary condition for designing a controller for self-adaptive systems, the controllability property is. Even if we do not have an explicit open loop model, a qualitative analysis should be performed.

Properties of the Closed Loop Model. When an explicit model of the open loop is available, the closed loop model can be synthesized mathematically to achieve the properties the designer aims to guarantee. In general, the controller is designed to achieve stability, robustness and performance.

Stability refers to whether control corrections move the open system state, over time, toward the reference value or level. A system is unstable if the controller causes overcorrections that never decrease, or that increase without limit, or that oscillates indefinitely. Instability can be introduced by making corrections that are too large in an attempt to achieve the reference level quickly. This leads to oscillating behaviors in which the system overshoots the reference value alternately to the high side and the low side.

Robust stability, or robustness, is a special type of stability in control theory. Robust stability means that the closed loop system is stable in the presence of external disturbances, model parameters and model structure uncertainties.

Performance is another important property in closed loop models and can be measured in terms of rise time, overshoot, settling time, steady error or accuracy [2, 4].

4.4 Research Challenges

In this section, we have briefly described and analyzed how control theory properties can be used to guide the realization of assurances in the engineering of self-adaptive systems. However, for this realization we identify a number of challenges that require further research work.

Control theory challenges. We argue that the concepts and principles of control theory and the assurances they provide, at least in abstract form, can be applied in the design of a large class of self-adaptation problems in software systems. Part of these problems correspond to scenarios in which it is possible to apply control theory *directly* to self-adaptive systems, that is, by

mathematically modeling the software system behavior, and applying control theory techniques to obtain desired properties, such as stability, performance and robustness. These properties automatically provide corresponding assurances about the controlled system behavior. However, there are still no clear guidelines about the limitations of control theory as directly applied to self-adaptive systems in the general case.

Another set of problems corresponds to scenarios where it is infeasible to build a reasonably precise mathematical model, but instead, it is possible to create an approximated operational or even qualitative model of the self-adaptive system behavior. In this case, the formal definitions and techniques of control theory may not apply directly, but understanding the principles of control theory can guide the sorts of questions the designer should answer and take care of while designing an self-adaptive system.

Many challenges on the application of feedback control to perpetual assurances in self-adaptive systems arise from the following research questions:

- How can we determine whether a given self-adaptive system will be stable?
- How quickly will the system respond to a change in the reference value? Is this fast enough for the application? Are there lags or delays that will affect the response time? If so, are they intrinsic to the system or can they be optimized?
- What are the constraints for external disturbances and how do they affect self-adaptive systems design?
- Can we design a robust controller to achieve robust stability or use adaptive or hierarchical control?
- How accurately and periodically shall the system track the reference value? Is this good enough for the application domain?
- How much resources will the system spend in tracking and adjusting the reference value? Can this amount be optimized? What is more important: to minimize the cost of resources or the time for adjusting the reference values? Can this tradeoff be quantified?
- How likely is that multiple control inputs are needed to achieve robust stability?

Modeling challenges. These challenges concern the identification of the control core phenomena (*e.g.*, system identification or sampling periods). The analysis of the system model should determine whether the “knobs” have enough power (command authority) to actually drive the system in the required direction. Many open research questions remain, for example:

- How do we model explicitly and accurately the relationship among system goals, adaptation mechanisms, and the effects produced by controlled variables?
- Can we design software systems having an explicit specification of what we want to assure with control-based approaches? Can we do it by focusing only on some aspects for which feedback control is more effective?
- Can we improve the use of control, or achieve control-based design, by connecting as directly as possible some real physics inside the software systems?

- How far can we go by modeling self-adaptive systems mathematically? What are the limitations?
- How can we ensure an optimal sampling rate over time? What is the overhead introduced by oversampling the underlying system?
- Can we control the sampling rate depending on the current state of the self-adaptive system?
- How can we ensure an optimal sampling rate over time? What is the overhead introduced by oversampling the underlying system?
- Can we control the sampling rate depending on the current state of the self-adaptive system?

Run-time validation and verification (V&V) challenges. Run-time V&V

tasks are crucial in scenarios where controllers based on mathematical models are infeasible. Nonetheless, performing V&V tasks (*e.g.*, using model checking) over the entire system—at run-time, to guarantee desired properties and goals, is often infeasible due to prohibitive computational costs. Therefore, other fundamental challenges for the assurance of self-adaptive systems arise from the need of engineering incremental and composable V&V tasks [41]. Some open research questions on the realization of run-time V&V are:

- Which V&V tasks can guarantee which control properties, if any, and to what extent?
- Are stability, accuracy, settling-time, overshoot and other properties composable (*e.g.*, when combining control strategies which independently guarantee them)?
- What are suitable techniques to realize the composition of V&V tasks?
- Which approaches can be borrowed from testing? How can these be reused or adjusted for the assurance of self-adaptive systems?
- Regarding incrementality: in which cases is it useful? How can incrementality be realized? How increments are characterized, and their relationship to system changes?

Control strategies design challenges. As mentioned earlier, uncertainty is one of the most challenging problems in assuring self-adaptive systems. Thus, it is almost impossible to design controllers that work well for all possible values of references or disturbances. In this regard, models at run-time as well as adaptive and hierarchical (*e.g.*, three-layer hierarchies) control strategies are of paramount importance to self-adaptive systems design [30]. Relevant research questions include:

- How to identify external disturbances that affect the preservation of desired properties? What about external disturbances affecting third party services?
- How do we model the influence of disturbances on desired properties?
- How to deal with complex reference values? In the case of conflicting goals, can we detect such conflicting goals *a priori* or *a posteriori*?
- Can we identify the constraints linking several goals in order to capture a more complex composite goal (reference value)?

- Feedback control may also help in the specification of viability zones for self-adaptive systems. In viability zones desired properties are usually characterized in terms of several variables. How many viability zones are required for the assurance of a particular self-adaptive software system? Does each desired property require an independent viability zone? How to manage trade-offs and possible conflicts among several viability zones?
- How to maintain the causal connection between viability zones, adapted system, and its corresponding V&V software artifacts?

5 Summary and Conclusions

In this section, we present the overall summary of the identified key research challenges for the provision of assurances for self-adaptive systems. Though the theme of assurances is quite specific, the exercise was not intended to be exhaustive. Amongst the several topics involved by the challenges on the provision of assurances when engineering self-adaptive systems, we have focused on three major topics: perpetual assurances, composition and decomposition of assurances, and assurances inspired by control theory. We now summarize the most important research challenges for each topic.

- *Perpetual Assurances*—provision of perpetual assurances during the entire lifecycle of a self-adaptive system poses three key challenges: how to obtain a better understanding of the nature of uncertainty in software systems and how it should be equated, how to monitor and quantify uncertainty, and how to derive and integrate new evidence.
- *Composing and Decomposing Assurances*—although assurance cases can be used to collect and structure evidence, the key challenge is how to compose and decompose evidence in order to build arguments. There are two reasons for that: first, there is the need to manipulate different types of evidence and their respective assumptions because of the uncertainty permeating self-adaptive systems; and second, whenever a system adapts, it is expected that its associated assurance cases adapt, preferably autonomously because of the reduced involvement of humans in managing a self-adaptive system. Another challenge is the need to provide overarching processes that would allow us to manage assurance cases, both during development time and run-time, since assurance cases are dynamic and should be updated whenever the system self-adapts.
- *Control Theory Assurances*—although synergies have been identified between control theory and self-adaptive systems, the challenge that remains is the definition of clear guidelines that would facilitate the direct application of control theory principles and practices into self-adaptive systems. As a result, adapted properties from control theory could be used as evidence for the provision of assurances. Since modelling is a key aspect in control systems, the challenge is to identify, in the context of self-adaptive systems, the variables to be monitored and controlled, suitable control strategies to model for each case,

and how to implement these models directly in the adaptation mechanisms to fulfil the goals of the system. In order to deal with uncertainties, and inspired by practical control systems, there is the need to consider hierarchical structures of controllers, which should be supported by models that should be adaptable (*i.e.*, adaptive control). Due to the dynamic aspects of self-adaptive software systems, there is a need to perform the verification and validation tasks in an incremental and composable way, both at design and run-time.

There are several aspects that permeate the identified research challenges, but uncertainty is a key factor in the provision of assurances for self-adaptive systems. For example, there is uncertainty associated with the generation and composition of evidence that is used to build assurance arguments. In some contexts the only way to deal with uncertainty is to make assumptions—for example, assumptions on the number and type of changes, assumptions about the context in which the system operates, and assumptions associated with the produced evidence. The validity of the assumptions need to be perpetually evaluated while providing assurances. How to manage assumptions considering the limited involvement by humans during run-time of self-adaptive systems is a research challenge.

The autonomous way in which the provision of assurances ought to be managed is also considered a research challenge. For that, a separate feedback control loop might be needed to perpetually collect, structure and analyze the evidence. The role of this feedback control loop is not directly related to the services provided by the system, but to the management of the assurance arguments that justify the ability of the system to provide its intended service and its associated quality. Although a feedback control loop is an excellent mechanism to handle uncertainty, it should be considered under a set of assumptions, which also need to be evaluated during run-time.

Furthermore, considering the complexity of the task at hand, processes should be incorporated into the feedback control loop in order to manage the perpetual provision of assurances, which should depend, for example, on the trust level required by the system, the kind of evidence that the system is able to generate, and how this evidence can be composed in order to build assurance arguments. If there are any changes in the evidence or its assumptions, the controller should automatically evaluate the validity of the assurance arguments. Equally, if trust levels associated with the system goals change, the controller should evaluate the arguments of the assurance case, and if required, new evidence ought to be generated and composed in order to rebuild assurance arguments.

The identified research challenges are specifically associated with the three topics related to the provision of assurances when engineering self-adaptive systems, which were addressed in this paper. These are challenges that our community must face because of the dynamic nature of self-adaptation. Moreover, the ever changing nature of these type of systems requires to bring uncertainty to the forefront of system design. It is this uncertainty that challenges the applicability of traditional software engineering principles and practices, but motivates the search for new approaches when developing, deploying, operating, evolving and decommissioning self-adaptive systems.

References

1. Ali, R., Griggio, A., Franzén, A., Dalpiaz, F., Giorgini, P.: Optimizing monitoring requirements in self-adaptive systems. In: Bider, I., Halpin, T., Krogstie, J., Nurcan, S., Proper, E., Schmidt, R., Soffer, P., Wrycza, S. (eds.) BPMDS/EMMSAD -2012. LNIP, vol. 113, pp. 362–377. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31072-0_25
2. Åström, K.J., Murray, R.M.: Feedback Systems. An iNtroduction for Scientists and Engineers (2008)
3. Åström, K., Wittenmark, B.: Adaptive Control. Addison-Wesley series in Electrical Engineering: Control Engineering. Addison-Wesley (1995)
4. Balzer, B., Litoiu, M., Müller, H., Smith, D., Storey, M.A., Tilley, S., Wong, K.: 4th international workshop on adoption-centric software engineering. In: Proceedings of the 26th International Conference on Software Engineering, ICSE 2004, pp. 748–749. IEEE Computer Society, Washington, DC (2004)
5. Blair, G., Bencomo, N., France, R.B.: Models@run.time. *IEEE Comput.* **42**, 22–27 (2009)
6. Blanchette, Jr., S.: Assurance cases for design analysis of complex system of systems software. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, April 2009
7. Bloomfield, R., Bishop, P.: Safety and assurance cases: past, present and possible future-an Adelard perspective. In: Dale, C., Anderson, T. (eds.) Making Systems Safer, pp. 51–67. Springer, London (2010). https://doi.org/10.1007/978-1-84996-086-1_4
8. Bloomfield, R., Peter, B., Jones, C., Froome, P.: ASCAD – Adelard Safety Case Development Manual. Adelard, 3 Coborn Road, London E3 2DA, UK (1998)
9. Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Engineering self-adaptive systems through feedback loops. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_3
10. Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G.: Dynamic qos management and optimization in service-based systems. *IEEE Trans. Softw. Eng.* **37**(3), 387–409 (2011)
11. Casanova, P., Garlan, D., Schmerl, B., Abreu, R.: Diagnosing architectural runtime failures. In: Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2013, pp. 103–112 (2013)
12. Casanova, P., Garlan, D., Schmerl, B., Abreu, R.: Diagnosing unobserved components in self-adaptive systems. In: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, pp. 75–84 (2014)
13. Checiu, L., Solomon, B., Ionescu, D., Litoiu, M., Iszlai, G.: Observability and controllability of autonomic computing systems for composed web services. In: Proceedings of the 6th IEEE International Symposium on Applied Computational Intelligence and Informatics, (SACI 2011), pp. 269–274. IEEE (2011)
14. Cheng, B.H.C., et al.: Software engineering for self-adaptive systems: a research roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_1

15. Cheng, B.H.C., et al.: Using models at runtime to address assurance for self-adaptive systems. In: Bencomo, N., France, R., Cheng, B.H.C., Aßmann, U. (eds.) Models@run.time. LNCS, vol. 8378, pp. 101–136. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08915-7_4
16. Cheng, S.W., Garlan, D., Schmerl, B., Sousa, J.a.P., Spitznagel, B., Steenkiste, P.: Using architectural style as a basis for self-repair. In: Bosch, J., Gentleman, M., Hofmeister, C., Kuusela, J. (eds.) Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture, 25–31 August 2002, pp. 45–59. Kluwer Academic Publishers (2002)
17. Dumont, G., Huzmezian, M.: Concepts, methods and techniques in adaptive control. IEEE American Control Conf. (ACC) **2**, 1137–1150 (2002)
18. Esfahani, N., Malek, S.: Uncertainty in self-adaptive software systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-adaptive Systems II. LNCS, vol. 7475, pp. 214–238. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_9
19. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: 33rd International Conference on Software Engineering (ICSE), pp. 341–350, May 2011
20. Filieri, A., Tamburrelli, G.: Probabilistic verification at runtime for self-adaptive systems. In: Cámaras, J., de Lemos, R., Ghezzi, C., Lopes, A. (eds.) Assurances for Self-Adaptive Systems. LNCS, vol. 7740, pp. 30–59. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36249-1_2
21. Garlan, D.: Software engineering in an uncertain world. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER 2010, pp. 125–128 (2010)
22. Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D.M.: Feedback Control of Computing Systems. Wiley (2004)
23. IBM Corporation: An Architectural Blueprint for Autonomic Computing. IBM Corporation, Technical report (2006)
24. Iftikhar, M.U., Weyns, D.: Activforms: active formal models for self-adaptation. In: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, pp. 125–134 (2014)
25. Gil de la Iglesia, D., Weyns, D.: Guaranteeing robustness in a mobile learning application using formally verified MAPE loops. In: Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2013, pp. 83–92 (2013)
26. Kelly, T.P.: Managing complex safety cases. In: Redmill, F., Anderson, T. (eds.) Current Issues in Safety-Critical Systems, pp. 99–115. Springer, London (2003). https://doi.org/10.1007/978-1-4471-0653-1_6
27. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Comput. **36**(1), 41–50 (2003)
28. Kit, M., Gerostathopoulos, I., Bures, T., Hnetyntka, P., Plasil, F.: An architecture framework for experimentations with self-adaptive cyber-physical systems. In: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, pp. 93–96 (2015)
29. Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. In: FOSE 2007: 2007 Future of Software Engineering, pp. 259–268. IEEE Computer Society, Washington, DC (2007)

30. de Lemos, R., et al.: Software engineering for self-adaptive systems: a second research roadmap. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-adaptive Systems II. LNCS, vol. 7475, pp. 1–32. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_1
31. Litoiu, M.: A performance analysis method for autonomic computing systems. ACM Trans. Auton. Adapt. Syst. **2**(1), 3 (2007)
32. Karsai, G., Ledeczi, A., Sztipanovits, J., Peceli, G., Simon, G., Kovacshazy, T.: An approach to self-adaptive software based on supervisory control. In: Laddaga, R., Shrobe, H., Robertson, P. (eds.) IWSAS 2001. LNCS, vol. 2614, pp. 24–38. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36554-0_3
33. Mahdavi-Hezavehi, S., Avgeriou, P., Weyns, D.: A classification framework of uncertainty in architecture-based self-adaptive systems with multiple quality requirements. In: Mistrik, I., Ali, N., Kazman, R., Grundy, J., Schmerl, B. (eds.) Managing Trade-Offs in Adaptable Software Architectures, pp. 45–77. Morgan Kaufmann, Boston (2017)
34. Murray, R.M.: Control in an Information Rich World: Report of the Panel on Future Directions in Control, Dynamics, and Systems. SIAM (2003)
35. Perez-Palacin, D., Mirandola, R.: Uncertainties in the modeling of self-adaptive systems: a taxonomy and an example of availability evaluation. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, ICPE 2014, pp. 3–14 (2014)
36. Redondo, R.P.D., Arias, J.J.P., Vilas, A.F.: Reusing verification information of incomplete specifications. In: Component-based Software Engineering Workshop, Lund, Sweden (2002)
37. Schmerl, B., et al.: Challenges in composing and decomposing assurances for self-adaptive systems. In: de Lemos, R., Garlan, D., Ghezzi, C., Giese, H. (eds.) Self-Adaptive Systems III. LNCS, vol. 9640, pp. 64–89. Springer, Heidelberg (2017)
38. Schmerl, B., CáMara, J., Gennari, J., Garlan, D., Casanova, P., Moreno, G.A., Glazier, T.J., Barnes, J.M.: Architecture-based self-protection: composing and reasoning about denial-of-service mitigations. In: HotSoS 2014: 2014 Symposium and Bootcamp on the Science of Security, 8–9 April 2014, Raleigh, NC, USA (2014)
39. Shibata, T., Fukuda, T.: Hierarchical intelligent control for robotic motion. IEEE Trans. Neural Networks **5**(5), 823–832 (1994)
40. Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A., Zadok, E.: Runtime verification with state estimation. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 193–207. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_15
41. Tamura, G., et al.: Towards practical runtime verification and validation of self-adaptive software systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-adaptive Systems II. LNCS, vol. 7475, pp. 108–132. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_5
42. Villegas, N., Tamura, G., Müller, H.: Chapter 2 - Architecting software systems for runtime self-adaptation: concepts, models, and challenges. In: Mistrik, I., Ali, N., Kazman, R., Grundy, J., Schmerl, B. (eds.) Managing Trade-Offs in Adaptable Software Architectures, pp. 17–43. Morgan Kaufmann, Boston (2017)
43. Villegas, N., Müller, H., Tamura, G., Duchien, L., Casallas, R.: A framework for evaluating quality-driven self-adaptive software systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011), pp. 80–89. ACM (2011)

44. Villegas, N.M., Tamura, G., Müller, H.A., Duchien, L., Casallas, R.: DYNAMICO: a reference model for governing control objectives and context relevance in self-adaptive software systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-adaptive Systems II. LNCS, vol. 7475, pp. 265–293. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_11
45. Vogel, T., Giese, H.: Model-driven engineering of self-adaptive software with EUREMA. ACM Trans. Auton. Adapt. Syst. **8**(4), 18:1–18:33 (2014)
46. Weyns, D., et al.: Perpetual assurances for self-adaptive systems. In: de Lemos, R., Garlan, D., Ghezzi, C., Giese, H. (eds.) Self-Adaptive Systems III. LNCS, vol. 9640, pp. 31–63. Springer, Heidelberg (2017)
47. Weyns, D., Calinescu, R.: Tele assistance: a self-adaptive service-based system exemplar. In: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-managing Systems, SEAMS 2015, pp. 88–92. IEEE Press, Piscataway (2015)
48. Weyns, D., Iftikhar, M.U., de la Iglesia, D.G., Ahmad, T.: A survey of formal methods in self-adaptive systems. In: Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering, C3S2E 2012, pp. 67–79(2012)
49. Weyns, D., Malek, S., Andersson, J.: FORMS: unifying reference model for formal specification of distributed self-adaptive systems. ACM Trans. Auton. Adapt. Syst. **7**(1), 8:1–8:61 (2012)
50. Ye, F., Kelly, T.: Contract-based justification for cots component within safety-critical applications. In: Cant, T. (ed.) Ninth Australian Workshop on Safety-Related Programmable Systems (SCS 2004). CRPIT, vol. 47, pp. 13–22. ACS, Brisbane (2004)
51. Zhang, J., Cheng, B.H.: Using temporal logic to specify adaptive program semantics. J. Syst. Softw. **79**(10), 1361–1369 (2006)

Perpetual Assurances for Self-Adaptive Systems

Danny Weyns^{1,2(✉)}, Nelly Bencomo³, Radu Calinescu⁴,
Javier Camara⁵, Carlo Ghezzi⁶, Vincenzo Grassi⁷, Lars Grunske⁸,
Paola Inverardi⁹, Jean-Marc Jezequel¹⁰, Sam Malek¹¹,
Raffaela Mirandola⁶, Marco Mori¹², and Giordano Tamburrelli¹³

¹ Katholieke Universiteit Leuven, Louvain, Belgium

danny.weyns@cs.kuleuven.be

² Linnaeus University Sweden, Växjö, Sweden

³ Aston University, Birmingham, UK

⁴ University of York, York, UK

⁵ Carnegie Mellon University, Pittsburgh, USA

⁶ Politecnico di Milano, Milan, Italy

⁷ University of Rome, Rome, Italy

⁸ Humboldt University of Berlin, Berlin, Germany

⁹ University of L'Aquila, L'Aquila, Italy

¹⁰ IRISA, Rennes, France

¹¹ University of Irvine, Irvine, USA

¹² University of Florence, Florence, Italy

¹³ Vrije Universiteit Amsterdam, Amsterdam, The Netherlands

Abstract. Providing assurances for self-adaptive systems is challenging. A primary underlying problem is uncertainty that may stem from a variety of different sources, ranging from incomplete knowledge to sensor noise and uncertain behavior of humans in the loop. Providing assurances that the self-adaptive system complies with its requirements calls for an enduring process spanning the whole lifetime of the system. In this process, humans and the system jointly derive and integrate new evidence and arguments, which we coined *perpetual assurances* for self-adaptive systems. In this paper, we provide a background framework and the foundation for perpetual assurances for self-adaptive systems. We elaborate on the concrete challenges of offering perpetual assurances, requirements for solutions, realization techniques and mechanisms to make solutions suitable. We also present benchmark criteria to compare solutions. We then present a concrete exemplar that researchers can use to assess and compare approaches for perpetual assurances for self-adaptation.

1 Introduction

In recent years, researchers have started studying the impact of self-adaptation on the software engineering process [1, 7, 42]. This led to the insight that today's iterative, incremental, and evolutionary software engineering processes do not meet the requirements of many contemporary systems that need to handle change during system operation. In self-adaptive systems change activities are shifted from development time

to runtime, and the responsibility for these activities is shifted from software engineers or system administrators to the system itself. Therefore the traditional boundary between development time and runtime blurs, which requires a complete reconceptualization of the software engineering process. An important aspect of the software engineering process of self-adaptive systems – in particular business or safety critical systems – is providing new evidence that the system requirements are satisfied during its entire lifetime, from inception to and throughout operation. This evidence must be produced despite the uncertainty that affects the requirements and/or the behavior of the environment, which lead to changes that may affect the system, its goals, and its environment [22, 27, 62]. To provide guarantees that the system requirements are satisfied, the state of the art in self-adaptive systems advocates the use of formal models as one promising approach, see e.g., [13, 31, 39, 63, 64, 70]. Some approaches employ formal methods to provide guarantees by construction. More recently, the use of probabilistic models to handle uncertainties has gained interest. These models are used to verify properties and support decision-making about adaptation at runtime. However, providing assurances that the goals of self-adaptive systems are achieved during the entire life cycle remains a difficult challenge.

In this paper, we provide a background framework and the foundation for an approach to providing assurances for self-adaptive systems that we coined “perpetual assurances for self-adaptive systems.” We elaborate on the challenges of perpetual assurances, requirements for solutions, realization techniques and mechanisms to make solutions suitable, and benchmark criteria to compare solutions. We then present a case study that researchers can use to assess, challenge and compare their approaches to providing assurances for self-adaptation. The paper concludes with a summary of the main challenges we identified to realize perpetual assurances for self-adaptive systems.

2 Key Challenges for Perpetual Assurances

We use the following working definition for *assurances for self-adaptive systems*:

Assurances for self-adaptive systems mean providing evidence for requirements compliance; this evidence can be provided off-line (i.e., not directly connected or controlled by the running system) and complemented online (i.e., connected or under control of the running system).

We use the following working definition for *perpetual assurances for self-adaptive systems*:

Perpetual assurances for self-adaptive systems mean providing evidence for requirements compliance through an enduring process that continuously provides new evidence by combining system-driven and human-driven activities to deal with the uncertainties that the system faces across its lifetime, from inception to and throughout operation in the real world.

Thus, providing assurances cannot be achieved by simply using off-line solutions possibly complemented with online solutions. Instead, we envisage that perpetual assurances will employ a continuous process where humans and the system jointly and continuously derive and integrate new evidence and arguments required to assure stakeholders (e.g., end users and system administrators) that the requirements are met by the self-adaptive system despite the uncertainties it faces throughout its lifetime.

Realizing the vision of perpetual assurances for self-adaptive systems poses multiple challenges, which we discuss next. From these challenges, we identify requirements, we overview relevant approaches for assurances and discuss mechanisms to make these approaches suitable for perpetual assurances, and finally we define benchmark criteria to compare different solutions.

A primary underlying challenge stems from *uncertainty*. The literature provides several definitions of uncertainty, ranging from *absence of knowledge* to *inadequacy of information* and *the difference between the amount of information required to perform a task and the amount of information already possessed* [5, 34, 49, 52, 59, 65]. [59] and later [51] refer to uncertainty as *any deviation from the unachievable ideal of completely deterministic knowledge of the relevant system*. Such deviations can lead to a lack of confidence in the obtained results, based on a judgment that they might be incomplete, blurred, inaccurate, unreliable, inconclusive, or potentially false [53].

Hereafter, we make use of the taxonomy of uncertainty initially proposed in [51], where uncertainties are classified along three dimensions: *location*, *level*, and *nature*. The *location* dimension refers to where uncertainty manifests in the description (the model) of the studied system or phenomenon. We can specialize it into: (i) input parameters, (ii) model structure, and (iii) context. The *level* dimension rates how uncertainty occurs along a spectrum ranging from deterministic knowledge (level 0) to total ignorance (level 3). Awareness of uncertainty (level 1) and unawareness (level 2) are the intermediate levels. The *nature* dimension indicates whether the uncertainty is due to the lack of accurate information (*epistemic*) or to the inherent variability of the phenomena being described (*aleatory*). Other taxonomies of uncertainties related to different modeling aspects can be found [8, 36, 54].

Recognizing the presence of uncertainty and managing it can mitigate its potentially negative effects and increase the level of assurance in a given self-adaptive system. By ignoring uncertainties, one could draw unsupported claims on the system validity or generalize them beyond their bounded scope.

Table 1 shows a classification of uncertainty based on [48], which is inspired by the literature on self-adaptation (e.g., [30, 35].) Each source of uncertainty is classified according to the *location* and *nature* dimensions of the taxonomy. The *level* dimension of the taxonomy is not shown in the table since each source of uncertainty can be of any level depending on the implemented capabilities in the system that should deal with the uncertainty.

Sources of uncertainty are structured in four groups: (i) sources of uncertainty related to the *system* itself; (ii) uncertainty related to the system *goals*; (iii) uncertainty in the execution *context*; and (iv) uncertainty related to *human* aspects. We provide a brief definition of each type of uncertainty source, focusing on *incompleteness*, *automated learning*, and *multiple ownership*, which especially apply to self-adaptive systems.

Sources of uncertainty related to the system itself:

- *Simplifying assumptions*: the model, being an abstraction, includes per-se a degree of uncertainty, since details whose significance is supposed to be minimal are not included in the model.

Table 1. Classification of sources of uncertainty based on [48]

Source of Uncertainty		Classification	
		Location	Nature
Simplifying assumptions	System	Structural/context	Epistemic
Model drift		Structural	Epistemic
Incompleteness		Structural	Epistemic/Aleatory
Future parameters value		Input	Epistemic
Adaptation functions		Structural	Epistemic/Aleatory
Automatic learning		Structural/input	Epistemic/Aleatory
Decentralization		Context/structural	Epistemic
Requirements elicitation	Goals	Structural/input	Epistemic/Aleatory
Specification of goals		Structural/input	Epistemic/Aleatory
Future goal changes		Structural/input	Epistemic/Aleatory
Execution context	Context	Context/structural/input	Epistemic
Noise in sensing		Input	Epistemic/Aleatory
Different sources of information		Input	Epistemic/Aleatory
Human in the loop	Human	Context	Epistemic/Aleatory
Multiple ownership		Context	Epistemic/Aleatory

- *Model drift*: as the system adapts, its structure also changes, so it is necessary to keep aligned systems and corresponding models to avoid reasoning on outdated models.
- *Incompleteness*: manifests when some parts of the system or its model are knowingly missing. Incompleteness can show up at design time and also at runtime. Progressive completion at design time occurs in iterative, exploratory development processes. Completion at runtime occurs, for example, in dynamic service discovery and binding in service-oriented systems. Managing incompleteness is challenging. Performing assurance on incomplete systems requires an ability to discover which sub-assurances must be delayed until completion occurs, and this may even happen at runtime [23]. Consequently, assurance must be incremental and comply with the time constraints imposed by the runtime environment.
- *Future parameter value*: uncertainty in the future world where the system will execute creates uncertainties in the correct actions to take at present.
- *Automatic learning*: in self-adaptive systems usually uses statistical processes to create knowledge about the execution context and most-useful behaviors. These learning processes can lead to applications with uncertain behavior. The location of this uncertainty may be in the model structure or input parameters, depending on how general the concepts are for which the application has been provided with machine-leaning capabilities. The nature of the uncertainty depends on the point of view. From the viewpoint of the application and its models of the world, as the machine has to learn from imperfect and limited data, the nature of the uncertainty is epistemic. From the user viewpoint, since the information undergoes a statistical learning process during model synthesis, automatic learning introduces randomness

in the model and analysis results, and consequently it can be seen as an aleatory uncertainty.

- *Decentralization*: this uncertainty comes from the difficulty to keep a model updated if the system is decentralized and each subsystem has self-adaptation capabilities. The reason is that it can be hardly expected that any subsystem could obtain accurate knowledge of the state of the entire system.

Sources of uncertainty related to the goals of the system:

- *Elicitation of requirements*: identifying the needs of stakeholders is known to be problematic in practice. Issues concerns defining the system scope, problems in fostering understanding among the different communities with a stake in the system, and problems in dealing with the volatile nature of requirements. The latter issue is particularly relevant for self-adaptation.
- *Specification of goals*: accurately specifying the preferences of stakeholders is difficult and prone to uncertainty. For example, the runtime impact of a utility function that defines the desired tradeoffs between conflicting qualities may not be completely known upfront.
- *Future goal changes*: the requirements of the system can change due to new needs of the customers, to new regulations or to new market rules, etc.

Sources of uncertainty related to the execution context of the system:

- *Execution context*: the context in which the application operates should be represented in the model, but the available monitoring mechanisms might not suffice to univocally determine this context and its evolution over time.
- *Noise in sensing*: the sensors/monitors are not ideal devices and they can provide, for example, slightly different data in successive measures while the actual value of the monitored data did not change.

Sources of uncertainty related to the humans aspects:

- *Human in the loop*: human behavior is intrinsically uncertain and systems commonly rely on correct human operations – both as system users and as system administrators. This assumption of correct operations may not hold because human behavior can diverge from the correct behavior.
- *Multiple ownership*: systems are increasingly assembled out of parts provided by different stakeholders, whose exact nature and behavior may be partly unknown when the system is composed. This is typical of service-oriented, dynamically composed systems, and systems of systems. The nature of this uncertainty is mainly epistemic. However, it may also be aleatory since third-party components may change without notifying the owner of the integrated application.

As discussed above, uncertainty in its various forms represents as the ultimate source of both motivations for and challenges to perpetual assurance. Uncertainty manifests through *changes*. For example, uncertainty in capturing the precise behavior of an input phenomenon to be controlled results in assumptions made to implement the system. Therefore, the system must be calibrated later when observations of the

physical phenomenon are made. This in turn leads to changes in the implemented control system that must be scrutinized for assurances.

3 Requirements for Solutions to Realize Perpetual Assurances

The provision of perpetual assurance for self-adaptive systems must satisfy additional requirements compared to those met by assurance solutions for non-adaptive systems. These additional requirements correspond to the challenges summarized in the previous section. In particular, perpetual assurance solutions must cope with a variety of uncertainty sources that depend on the purpose of self-adaptation and the environment in which the assured self-adaptive system operates. To this end, they must build and continually update their assurance arguments through integrating two types of assurance evidence. The first type of evidence corresponds to system and environment elements not affected significantly by uncertainty, and therefore can be obtained using traditional offline approaches [45]. The second type of evidence is associated with the system and environment components affected by the sources of uncertainty summarized in Table 1. These requirements basically refer to the different functions of adaptation: from sensing to monitoring, analyzing, planning, executing, and activating [23]. The evidence must be synthesized at runtime, when the uncertainty is treated, i.e., be reduced, quantified, or resolved sufficiently, to enable such synthesis. The requirements described below stem from this need to treat uncertainty, to generate new assurance evidence, and to use it to update existing assurance arguments.

Requirement 1: *A perpetual assurance solution must continually observe the sources of uncertainty affecting the self-adaptive system.* Without collecting data about the relevant sources of uncertainty, the solution cannot acquire the knowledge necessary to provide the assurance evidence unachievable offline.¹ Addressing this requirement may necessitate, for instance, the instrumentation of system components, the sensing of environmental parameters, and the monitoring of user behavior.

Requirement 2: *A perpetual assurance solution must use its observations of uncertainty sources to continually quantify and potentially reduce the uncertainties affecting its ability to provide assurance evidence.* The raw data obtained through observing the sources of uncertainty need to be used to reduce the uncertainty (e.g., through identifying bounds for the possible values of an uncertain parameter). This uncertainty quantification and potential reduction must be sufficient to enable the generation of the assurance evidence that could not be obtained offline. For instance, rigorous learning techniques such as those introduced in [15, 17, 28, 29] can be used to continually update formal models whose runtime analysis generates new assurance evidence.

Requirement 3: *A perpetual assurance solution must continually deal with overlapping sources of uncertainty, and may need to treat these sources in a composeable*

¹ If all necessary assurance evidence can be derived using knowledge available offline, then we assume that the system can be implemented as a non-adaptive system.

fashion. The sources of uncertainty from Table 1 hardly ever occur apart or independently from each other. Instead, their observed effects are often compounded, consequently inhibiting the system from clearly assessing the extent to which it satisfies its requirements. Therefore, solutions, assurance arguments and the evidence they build upon need to be able to tackle composed sources of uncertainty.

Requirement 4: *A perpetual assurance solution must continually derive new assurance evidence.* Up-to-date knowledge that reduces the uncertainty inherent within the self-adaptive system and its environment must be used to infer the missing assurance evidence (e.g., to deal with model drift or handle goal changes as described in Sect. 2). As an example, new evidence can be produced through formal verification of updated models that reflect the current system behavior [11–14, 31]. Complementary approaches to verification include online testing, simulation, human reasoning and combinations thereof.

Requirement 5: *A perpetual assurance solution must continually integrate new evidence into the assurance arguments for the safe behavior of the assured self-managing system.* Guaranteeing safe behavior requires the provision of assurance arguments that, in the case of self-adaptive systems, must combine offline evidence with new evidence obtained online, once uncertainties are resolved to the extent that enables the generation of the new evidence.

Requirement 6: *A perpetual assurance solution may need to continually combine new assurance evidence synthesized automatically and provided by human experts.* Developing assurance arguments is a complex process that is carried out by human experts and includes the derivation of evidence through both manual and automated activities. Online evidence derivation will in general require a similar combination of activities, although it is conceivable that self-adaptive systems affected by reduced levels and sources of uncertainty might not always be able to assemble all the missing evidence automatically.

Requirement 7: *A perpetual assurance solution must provide assurance evidence for the system components, the human activities and the processes used to meet the previous set of requirements.* As described above, perpetual assurances for self-adaptive systems cannot be achieved without employing system components and a mix of automated and manual activities not present in the solutions for assuring non-adaptive systems. Each of these new elements will need to be assured either by human-driven or machine-driven approaches. As an example, a model checker used to obtain new assurance evidence arguments at runtime will need to be certified for this type of use.

In addition to the functional requirements summarized so far, the provision of perpetual assurance must be timely, non-intrusive and auditable, as described by the following non-functional requirements.

Requirement 8: *The activities carried out by a perpetual assurance solution must produce timely updates of the assurance arguments.* The provision of assurance arguments and of the evidence underpinning them is time consuming. Nevertheless, perpetual assurance solutions need to complete updating the assurance arguments

guaranteeing the safety of a self-adaptive system timely, before the system engages in operations not covered by the previous version of the assurance arguments.

Requirement 9: *The activities of the perpetual assurance solution, and their overheads must not impact the operation of the assured self-adaptive system.* The techniques employed by a perpetual assurance solution (e.g., instrumentation, monitoring and online learning, and verification) must not interfere with the ability of the self-adaptive system to deliver its intended functionality effectively and efficiently.

Requirement 10: *The assurance evidence produced by a perpetual assurance solution and the associated assurance arguments must be auditable by human*

Table 2. Summary requirements

Requirement	Brief description
R1: Monitor uncertainty	A perpetual assurance solution must continually observe the sources of uncertainty affecting the self-adaptive system
R2: Quantify uncertainty	A perpetual assurance solution must use its observations of uncertainty sources to continually quantify and potentially reduce the uncertainties affecting its ability to provide assurance evidence
R3: Manage overlapping uncertainty sources	A perpetual assurance solution must continually deal with overlapping sources of uncertainty and may need to treat these sources in a composable fashion
R4: Derive new evidence	A perpetual assurance solution must continually derive new assurance evidence arguments
R5: Integrate new evidence	A perpetual assurance solution must continually integrate new evidence into the assurance arguments for the safe behavior of the assured self-managing system
R6: Combine new evidence	A perpetual assurance solution may need to continually combine new assurance evidence synthesized automatically and provided by human experts
R7: Provide evidence for the elements and activities that realize R1–R6	A perpetual assurance solution must provide assurance evidence for the system components, the human activities, and the processes used to meet the previous set of requirements
R8: Produce timely updates	The activities carried out by a perpetual assurance solution must produce timely updates of the assurance arguments
R9: Limited overhead	The activities of the perpetual assurance solution and their overheads must not impact the operation of the assured self-adaptive system
R10: Auditable arguments	The assurance evidence produced by a solution and the associated assurance arguments must be auditable by human stakeholders

stakeholders. Assurance arguments and the evidence they build upon are intended for human experts responsible for the decision to use a system, for auditing its safety, and for examining what went wrong after failures. In keeping with this need, perpetual assurance solutions must express their new assurance evidence in human-readable format.

Table 2 summarizes the requirements for perpetual assurance solutions.

4 Approaches to Realize Perpetual Assurances

Several approaches have been developed in the previous decades to check whether a software system complies with its requirements. Important relevant research in this context is models at runtime [4, 9]. In this section, we give a brief overview of representative approaches. In the next section, we elaborate on mechanisms to make these approaches suitable to support perpetual assurances for self-adaptive systems, driven by the requirements described in the previous section. Table 3 gives an overview of the approaches we discuss, organized in three groups: human-driven approaches (manual), system-driven (automated), and hybrid (manual and automated).

4.1 Human-Driven Approaches

We discuss two representative human-driven approaches for assurances: formal proof, and simulation.

Table 3. Approaches for assurances

Group	Approaches
Human-driven approaches	- Formal proof - Simulation
System-driven approaches	- Runtime verification - Sanity checks - Contracts
Hybrid approaches	- Model checking - Testing

Formal proof is a mathematical calculation to prove a sequence of connected theorems or formal specifications of a system. Formal proofs are conducted by humans with the help of interactive or automatic theorem provers, such as Isabelle or Coq. Formal proofs are rigorous and unambiguous, but require from the user both detailed knowledge about how the self-adaptive system works and significant mathematical experience. Furthermore, formal proofs only work for highly abstract models of a self-adaptive system (or more detailed models of small parts of it). Still the approach can be useful to build up assurance evidence arguments about basic properties of the system or specific elements (e.g., algorithms). We give two examples of formal proof used in self-adaptive systems. In [66], the authors formally prove a set of theorems to assure atomicity of adaptations of business processes in cross-organizational collaborations. The approach is illustrated in an example where a supplier wants to evolve its

business process by changing the policy of payments. In [69, 70], the authors formally prove a set of theorems to assure safety and liveness properties of self-adaptive systems. The approach is illustrated for data stream elements that modify their behavior in response to external conditions through the insertion and removal of filters.

Simulation comprises three steps: the design of a model of a system, the execution of that model, and the analysis of the output. Different types of models at different levels of abstraction can be used, including declarative, functional, constraint, spatial or multi-model. A key aspect of the execution of a model is the way time is treated, e.g., small time increments or progress based on events. Simulation allows a user to explore many different states of the system, without being prevented by an infinite (or at least very large) state space. Simulation runs can provide assurance evidence arguments at different levels of fidelity, which are primarily based on the level of abstraction of the model used and the type of simulation applied. As an example, in [19] the authors simulate self-adaptive systems and analyze the gradual fulfillment of requirements. The approach provides valuable feedback to engineers to iteratively improve the design of self-adaptive systems.

4.2 System-Driven Approaches

We discuss three representative system-driven approaches for assurances: runtime verification, sanity checks, and contracts.

Runtime verification is a well-studied lightweight verification technique that is based on extracting information from a running system to detect whether certain properties are violated. Verification properties are typically expressed in trace predicate formalisms, such as finite state machines, regular expressions, and linear temporal logics. Runtime verification is less complex than traditional formal verification approaches, such as model checking, because only one or a few execution traces are analyzed. As an example, in [54] the authors introduce an approach to estimate the probability that a temporal property is satisfied by a run of a program. The approach models event sequences as observation sequences of a Hidden Markov Model (HMM) and uses an algorithm for HMM state estimation, which determines the probability of a state sequence.

Sanity checks are calculations that check whether certain invariants hold at specific steps in the execution of the system. Sanity checks have been used for decades because they are very simple to implement and can easily be disabled when needed to address performance concerns. On the other hand, sanity checks provide only limited assurance evidence arguments that the system is not behaving incorrectly. As an example, in [56], the authors present an approach to manage shared resources that is based on an adaptive arbitrator that uses device and workload models. The approach uses sanity checks to evaluate the correctness of the decisions made by a constraint solver.

Contracts. The notion of Contract (introduced by B. Meyer in Design by Contract in Eiffel) makes the link between simple sanity checks and more formal, precise and verifiable interface specifications for software components. Contracts extend the ordinary definition of abstract data types with pre-conditions, post-conditions and

invariants. Contracts can be checked at runtime as sanity checks, but can also serve as a basis for assume-guarantee modular reasoning. In [10] the authors have shown that the notion of functional contract as defined in Eiffel can be articulated in four layers: syntactic, functional, synchronization and quality of service contracts. For each layer, both runtime checks and assume-guarantee reasoning are possible.

4.3 Hybrid Approaches

We discuss two representative hybrid approaches to provide assurances: model checking and testing.

Model checking is an approach allowing designers to check that a property (typically expressed in some logic) holds for all reachable states in a system. Model checking is typically run offline and can only work in practice on either a high level abstraction of an adaptive system or on one of its components, provided it is simple enough. For example, in [25], the authors model MAPE loops of a mobile learning application in timed automata and verify robustness requirements expressed in timed computation tree logic using the Uppaal tool. The models and properties are concrete instances of a set of reusable formal templates for specifying MAPE-K loops [26]. To deal with uncertainty at design time and the practical coverage limitations of model checking for realistic system, several researchers have studied the application of model checking techniques at runtime. E.g., in [13] QoS requirements of service-based systems are expressed as probabilistic temporal logic formulae, which are then automatically analyzed at runtime to identify and enforce optimal system configurations. The approach in [41] applies model checking techniques at runtime with the aim of validating compliance with requirements (expressed in LTL) of evolving code artifacts.

Testing allows designers to check that the system performs as expected for a finite number of situations. Inherently testing cannot guarantee that a self-adaptive system fully complies with its requirements. Testing is traditionally performed before deployment. As an example, in [18] the authors introduce a set of robustness tests, which provide mutated inputs to the interfaces between the controller and the target system, i.e. probes. The set of robustness tests are automatically generated by applying a set of predefined mutation rules to the messages sent by probes. Traditional testing allows demonstrating that a self-adaptive system is capable of satisfying particular requirements before deployment. However, it cannot deal with unanticipated system and environmental conditions. Recently, researchers in self-adaptive systems have started investigating how testing can be applied at runtime to deal with this challenge. In [33], the authors motivate the need and identify challenges for the testing of self-adaptive systems at runtime. The paper introduces the so-called MAPE-T feedback loop for supplementing testing strategies with runtime capabilities based on the monitoring, analysis, planning, and execution architecture for self-adaptive systems.

5 Mechanisms to Make Perpetual Assurances Working

Turning the approaches of perpetual assurances into a working reality requires tackling several issues. The key challenge we face is to align the approaches with the requirements we have discussed in Sect. 3. For the functional requirements (R1 to R7), the central problem is how to collect assurance evidence arguments at runtime and compose them with the evidence acquired throughout the lifetime of the system possibly by different approaches. For the quality requirements (R8 to R10) the central problem is to make the solutions efficient and support the interchange of evidence arguments between the system and users. In this section, we first elaborate on the implications of quality requirements on approaches for perpetual assurances. Then we discuss two classes of mechanisms that can be used to provide the required functionalities for perpetual assurances and meet the required qualities: decomposition mechanisms and model-driven mechanisms.

5.1 Quality Properties for Perpetual Assurances Approaches

The efficiency of the approaches for perpetual assurances must be evaluated with respect to the size of the self-adaptive system (e.g., number of components in the system) and the dynamism it is subject to (e.g., the frequency of change events that the system has to face). An approach for perpetual assurances of self-adaptive systems is efficient if, for systems of realistic size and dynamism, it is able to:

- Provide results (assurances) within defined time constraints (depending on the context of use);
- Consume a limited amount of resources, so that the overall amount of consumed resources over time (e.g. memory size, CPU, network bandwidth, energy, etc.) remains within a limited fraction of the overall resources used by the system.
- Scale well with respect to potential increases in size of the system and the dynamism it is exposed to.

In the next sections, we discuss some promising mechanisms to make approaches for perpetual assurances work. Before doing so, it is important to note that orthogonal to the requirements for perceptual assurance in general, the level of assurance that is needed depends on the requirements of the self-adaptive system under consideration. In some cases, combining regular testing with simple and time-effective runtime techniques, such as sanity checks and contract checking, will be sufficient. In other cases, more powerful approaches are required. For example, model checking could be used to verify a safe envelope of possible trajectories of an adaptive system at design time, and verification at runtime to check whether the next change of state of the system keeps it inside the pre-validated envelope.

5.2 Decomposition Mechanisms for Perpetual Assurances Approaches

The first class of promising mechanisms for the perpetual provisioning of assurances is based on the principle of decomposition, which can be carried out along two dimensions:

1. Time decomposition, in which:
 - a. Some preliminary/intermediate work is performed off-line, and the actual assurance is provided on-line, building on these intermediate results;
 - b. Assurances are provided with some degree of approximation/coarseness, and can be refined if necessary.
2. Space decomposition where verification overhead is reduced by independently verifying each individual component of a large system, and then deriving global system properties through verifying a composition of its component-level properties. Possible approaches that can be used to this end are:
 - a. Flat approaches, that only exploit the system decomposition into components;
 - b. Hierarchical approaches, where the hierarchical structure of the system is exploited;
 - c. Incremental approaches targeted at frequently changing systems, in which re-verification are carried out only on the minimal subset of components affected by a change.

We provide examples of state of the art approaches for each type.

5.2.1 Time Decomposition

We consider two types of time decomposition: (i) partially offline, assurances provided online, and (ii) assurances with some degree of approximation.

Time decomposition: partially offline, assurances provided online. In [20, 31], the authors propose approaches based on the pre-computation at design time of a formal and parameterized model of the system and its requirements; the model parameters are then instantiated at runtime, based on the output produced by a monitoring activity. Parametric probabilistic model checking [38] enables the evaluation of temporal properties on Parametric Markov Chains, yielding polynomials or rational functions as evaluation results. Instantiating the parameters in the result function, results for quantitative properties, such as probabilities or rewards can be cheaply obtained at runtime, based on a more computationally costly analysis process carried out offline. Due to limitations in practice of the parametric engines implemented in tools such as PARAM [38] or PRISM [44], the applicability of this technique may provide better results in self-adaptive systems with static architectures (or with a limited number of possible configurations), since dynamic changes to the architecture might require the generation of parametric expressions for new configurations.

Time decomposition: assurances with some degree of approximation. Statistical model checking [37, 47, 67] is a kind of simulation approach that enables the evaluation of probability and reward-based quantitative properties, similarly to probabilistic model checking. Specifically, the approach involves simulating the system for finitely many executions and using statistical hypothesis testing to determine whether the samples constitute statistical evidence of the satisfaction of a probabilistic temporal logic specification, without requiring the construction of an explicit representation of the state space in memory. Moreover, this technique provides an interesting tradeoff between analysis time and the accuracy of the results obtained (e.g., by controlling the number of sample system executions generated) [68]. This makes it appropriate for

systems with strict time constraints in which accuracy is not a primary concern. Moreover, dynamic changes to the system architecture do not penalize resource consumption (e.g., no model reconstruction is required), making statistical model checking a good candidate for the provision of assurances in highly dynamic systems. Initial research that uses statistical techniques at runtime for providing guarantees in self-adaptive systems is reported in [40, 61].

5.2.2 Space Decomposition

We consider three types of space decomposition: (i) flat approaches, (ii) hierarchical approaches, and (iii) incremental approaches.

Space decomposition: flat approaches. In [45], the authors use assume-guarantee model checking to verify component-based systems one component at a time, and employs regular safety properties both as the assumptions made about system components and the guarantees they provide to reason about probabilistic systems in a scalable manner. [50] employs probabilistic interface automata to enable the isolated analysis of probabilistic properties in environment models (based on shared system/environment actions), which are preserved in the composition of the environment and a non-probabilistic model of the system.

Space decomposition: hierarchical approaches. The authors of [43, 46] consider the decomposition of models to carry out incremental quantitative verification. This strategy enables the reuse of results from previous verification runs to improve efficiency. While [46] considers decomposition of Markov Decision Process-based Models into their strongly connected components (SCCs), [43] employs high-level algebraic representations of component-based systems to identify and execute the minimal set of component-wise re-verification steps after a system change.

Space decomposition: incremental approaches. In [16], the authors use assume-guarantee compositional verification to efficiently re-verify safety properties after changes that match some particular patterns, such as component failures.

5.2.3 Discussion

The discussed decomposition mechanisms lend themselves to the identification of possible division of responsibilities between human-driven and system-driven activities, towards the ultimate goal of providing guarantees that the system goals are satisfied. As an example, considering time decomposition approaches based on parametric model checking, the adopted parametric models can result from a combination of human-driven activities and artifacts automatically produced from other sources or information (architectural models, adaptation strategy specifications, etc.). Similarly, in case of space decomposition approaches based on assume-guarantee compositional verification, the assumptions used in the verification process could be automatically “learned” by the system [55], or provided by (or combined with input from) human experts.

5.3 Model-Based Mechanisms for Perpetual Assurances Approaches

For whatever division of responsibilities between human and systems in the perpetual assurance process, an important issue is how to define in a clean, well-documented and traceable way the interplay between the actors involved in the process. Model-driven mechanisms can be useful to this end, as they can support the rigorous development of a self-adaptive system from its high-level design up to its running implementation. Moreover, they can support the controlled and traceable modification by humans of parts of the system and/or of its self-adaptive logic, e.g. to respond to modifications of the requirements to be fulfilled. In this direction [58] presents a model-driven approach to the development of adaptation engines. This contribution includes the definition of a domain-specific language for the modeling of adaptation engines, and a corresponding runtime interpreter that drives the adaptation engine operations. Moreover, the approach supports the combination of on-line machine controlled adaptations and off-line long-term adaptations performed by humans to maintain and evolve the system.

Steps towards the definition of mechanisms that can support the human-system interplay in the perpetual assurance process can also be found in [39]. The authors propose an approach called ActivFORMS in which a formal model of the adaptation engine (MAPE-K feedback loop) based on timed automata and adaptation requirements expressed in timed computation tree logic are complemented by a suitable virtual machine that can execute the models, hence guaranteeing at runtime the compliance of properties verified offline. The approach allows dynamically checking the adaptation requirements, complementing the evidence arguments derived from human-driven offline activities with arguments provided online by the system that could not be obtained offline (e.g., for performance reasons). The approach supports deployment of new models at runtime elaborated by human experts to deal with new goals.

6 Benchmark Criteria for Perpetual Assurances

This section provides benchmark criteria to compare different approaches that provide perpetual assurances. We identify benchmark criteria along four aspects: capabilities of approaches to provide assurances, basis of evidence for assurances, stringency of assurances, and performance of approaches to provide assurances. The criteria cover both functional and quality requirements for perpetual assurances techniques.

6.1 Capabilities of Approaches to Provide Perpetual Assurances

The first benchmark aspect compares the extent to which approaches differ in their ability to provide assurances for the requirements (goals) of self-adaptation. We distinguish the following benchmark criteria: variability (in requirements and the system itself), inaccuracy & incompleteness, conflicting criteria, user interaction, and handling alternatives.

Variability. Support for variability of requirements is necessary in ever-changing environments where requirements are not statically fixed, but are runtime entities that should be accessible and modifiable. An effective assurance approach should take into

account requirements variations, including addition, updating, and deletion of requirements. For instance, variations to service-level agreement (SLA) contract specifications may have to be supported by an assurance approach. Variations may also involve the system itself, in term of unmanaged modifications; that is, an assurance approach should support system variability in terms of adding, deleting, and updating managed system components and services. This process may be completely automatic or may involve humans.

Inaccuracy and incompleteness. The second benchmark criterion deals with the capability of an assurance approach to provide evidence with models that are not 100% precise and complete. Inaccuracy may refer to both system and context models. For instance, it may not be possible to accurately define requirements at design time, thus making it necessary to leave a level of uncertainty within the requirements. On the other hand context models representing variations to context elements may be inaccurate, which may affect the results of an assurance technique. Similarly, incompleteness may either concern system or context models. For instance, either new services or new context elements may appear only at runtime. An assurance technique should provide the best possible assurance level regarding inaccurate and incomplete models. Additionally, an assurance approach may support re-evaluation at runtime when the uncertainty related to inaccuracy and incompleteness is solved (at least partially).

Competing criteria. Assurance approaches should take into account possible competing criteria. While it is desirable to optimize the utility of evidence (coverage, quality) provided by an assurance approach, it is important that the approach allows making a tradeoff between the utility of the evidence it provides and the costs for it (time, resource consumption).

User interaction. Assurance is related also to the way users interact with the system, in particular with respect to changing request load and changing user profiles/preferences. For instance, when the request load for a service increases, the assured level of availability of that service may be reduced. Self-adaptation may reconfigure the internal server structure (i.e., adding new components) to maintain the required level of assurance. In the case of changing user preference/profile variations, the level of assurances may either have to be recomputed or they may be interpreted differently. An assurance technique should be able to support such variations in the ways the system is used.

Handling alternatives is another important benchmark criterion for assurance of self-adaptive systems. An assurance technique should be able to deal with a flat space of reconfiguration strategies but also with one obtained by composition. In this context an important capability is to support assurances in the case one adaptation strategy is pre-empted by another one.

6.2 Basis of Assurance Benchmarking

The second aspect of benchmarking techniques for perpetual assurances defines the basis of assurance, i.e., the reasons why the researchers believe that the technique

makes valid decisions. We consider techniques based on historical data only, projections in the future only, and combined approaches. In addition, we consider human-based evidence as a basis for assurance.

Historical data only. At one extreme, an adaptation decision can be solely based on the past historical data, e.g., [12, 17, 28, 29, 32, 61]. An issue with such adaptive systems is that there may be no guarantees that the analyses based on historical data will be reified and manifested in the future executions of system. Nevertheless, in settings where the past is a good indicator of the system's future behavior, such approaches could be applicable.

Projections into the future only. At the other extreme, a system may provide assurances that project into the future, meaning that the analysis is based on predictive models of what may occur in the future. The predictions may take different forms, including statistical methods e.g., ARIMA-based forecasting approaches [2, 3, 21] and fuzzy methods to address the uncertainty with which different situations may occur [30, 36, 49]. A challenge with this approach is the difficulty of constructing predictive models.

Combined approaches. Finally, in many adaptive systems, combinations of the aforementioned techniques are used, i.e., some past historical data is used together with what-if analysis of the implications of adaptations over the predicted behaviors of the system in its future operation.

Human-based evidence. Since users may be involved in the assurance process, we consider human-based evidence as a further basis for assurance. Input from users may be required before performing adaptations, which may be critical in terms of provided services. Consider a situation in which a self-adaptive service-oriented system needs to be reconfigured by replacing a faulty service. Before substituting this service with a new one, selected by an automated assurance technique, the user may want to check its SLA to confirm its suitability as a replacement according to a criterion that cannot be easily applied automatically. Such a criterion may be subjective, not lending itself to quantification. Thus, by means of accepting the SLA of the new service, the user provides a human-based evidence to complement the assurances provisioned through an automated technique, which can be based on past historical data, predictive models or a combination of both.

Discussion. When publishing research results, it is important to explicitly discuss and benchmark the basis of assurances. For instance, if historical data is used for making decisions, the research needs to assess whether past behavior is a good indicator of future behavior, and if so, quantify such expectations. Similarly, if predictive models are used for making decisions, the researchers need to assess the accuracy of the predictions, ideally in a quantitative manner.

6.3 Stringency of Perpetual Assurances

A third important benchmarking aspect of techniques for perpetual assurances of self-adaptive systems is the nature of rationale and evidence of the assurances, which

we term “stringency of assurances.” The rationale for assurances may differ greatly depending on the purpose of the adaptive system and its users. While for one system the only proper rationale is a formal proof, for another system, proper rationale may be simply statistical data, such as the probability with which the adaptation has achieved its objective in similar prior situations. For example, a safety-critical system may require a more stringent assurance rationale than a typical consumer software system. Note that the assurance rationale is a criterion that applies for different phases of adaptation (monitoring, analysis, planning, and execution). Different phases of the same adaptive system may require different level of assurance rationale. For instance, while a formal proof is a plausible rationale for the adaptation decisions made during the planning phase, it may not be appropriate for the monitoring phase. Similarly, confidence intervals for the distribution of monitored system parameters may be a plausible rationale for the monitoring phase, but not necessarily for the execution phase of a self-adaptive software system.

6.4 Performance of Approaches to Provide Perpetual Assurances

Finally, for an assurance approach to be applicable in real-world scenarios it is essential to evaluate its performance, which is the fourth and final proposed benchmark aspect. We consider a set of benchmarks, which directly follow from efficiency and scalability concerns discussed in Sect. 4, namely timeliness, computational overhead, and complexity analysis.

Timeliness. The time required to achieve the required evidence is a key performance benchmark criteria, which is obvious for techniques that are used at runtime.

Computational overhead. Related to timeliness is the computational overhead, i.e., the consumed resources (e.g., memory and CPU) for enacting the assurance approach.

Complexity analysis. As each assurance technique has an (implicit) complexity that may affect its applicability in terms of the analysis required for a problem at hand, we propose a complexity analysis benchmark. Assurance techniques may be benchmarked in terms of their scope of applicability across different types of problems.

The following table summarizes the benchmark aspects, and for each of them the different benchmark criteria.

Some of the benchmark criteria from Table 4 directly reflect a specific requirement from Table 2. For example, the ‘Timeliness’ criterion is directly linked to requirement R8 (‘Produce timely updates’). Other criteria relate to multiple requirements from Table 2. As an example, the ‘Human evidence’ criterion links to R5 (‘Integrate new evidence’), R7 (‘Provide evidence for human activities that realize R5’), and R10 (‘Auditable arguments’). Finally, some arguments only link indirectly to the requirements from Table 2. This is the case for the ‘Handling alternatives’ criterion, which relates to the solution for self-adaptation and this solution may provide different levels of support for the requirements of perpetual assurances.

Table 4. Summary benchmark aspects and criteria

Benchmark Aspect	Benchmark Criteria	
	Criteria	Description
Capabilities of approaches to provide assurances	Variability	Capability of an approach to handle variations in requirements (adding, updating, deleting goals), and the system (adding, updating, deleting elements)
	Inaccuracy & incompleteness	Capability of an approach to handle inaccuracy and incompleteness of models of the system and context
	Competing criteria	Capability of an approach to balance the tradeoffs between utility (e.g., coverage, quality) and cost (e.g., time, resources)
	User interaction	Capability of an approach to handle changes in user behavior (preferences, profile)
	Handling alternatives	Capability of an approach to handle changes in adaptation strategies (e.g., pre-emption)
Basis of assurance benchmarking	Historical data only	Capability of an approach to provide evidence over time based on historical data
	Projections in the future	Capability of an approach to provide evidence based on predictive models
	Combined approaches	Capability of an approach to provide evidence based on combining historical data with predictive models
	Human evidence	Capability of an approach to complement automatically gathered evidence by evidence provided by humans
Stringency of assurances	Assurance rational	Capability of the approach to provide the required rational of evidence for the purpose of the system and its users (e.g., completeness, precision)
Performance of approaches	Timeliness	The time an approach requires to achieve the required evidence
	Computational overhead	The resources required by an approach (e.g., memory and CPU) for enacting the assurance approach
	Complexity	The scope of applicability of an approach to different types of problems

7 Example Case

We now present a case that supports the evaluation, comparison, and ranking of approaches for perpetual assurances. The example case has been conceived to challenge the capability of approaches for self-adaptation to achieve perpetual provisioning of assurances for different requirements driven by the benchmark criteria discussed in

the previous section. The particular ways in which different approaches solve the challenges proposed by the example case offers two distinct advantages. First it allows comparing prototypal research efforts, and second it acts as a testbed to demonstrate the effectiveness of the different techniques adopted by the different self-adaptive solutions.

We start with motivating the domain of the case and provide a set of general adaptation scenarios. The scenarios are linked to the types of uncertainty occurring in self-adaptive systems and the requirements for perpetual assurances. Next we describe the concrete case. We then give generic adaptation scenarios and explain benchmark criteria. Finally, we illustrate a comparison of two approaches for a concrete scenario.

7.1 Domain and General Adaptation Scenarios

The case comes from the domain of service-based systems, in which software services offered by third-party providers are dynamically composed at runtime to deliver complex functionality. Service-based systems are widely used in e-commerce, online banking, e-health and many other types of applications. They increasingly rely on self-adaptation to cope with the uncertainties that are often associated with third-party services [6, 13, 14, 20, 61] as the loose coupling of service-oriented architectures makes online reconfiguration feasible. Hence, the example case is a prototypical application.

A typical service-based system consists of a composition of web services that are accessed remotely through a software application called a workflow engine. Several providers may offer services that provide the same functionality, often with different levels of performance, reliability, costs, etc. Service providers can register concrete services at the service registry. The workflow of a composite service finds addresses (end points) of concrete services via the service registry. The way in which a workflow engine employs concrete services in order to provide the functionality required by the user is specified in the workflow that the engine is executing.

Table 5 lists a set of generic adaptation scenarios for service-based systems. These scenarios provide increasing challenges to self-adaptation in general and the provision of perpetual assurances in particular. They are not meant to be exhaustive, but cover typical types of adaptation problems with different types of uncertainties.

Adaptation scenarios differ based on the characteristics summarized below.

Types of uncertainties that the system needs to handle: Different types of uncertainty present different challenges to adaptation solutions and to the approaches used for perpetual assurances; for example monitoring or quantifying uncertainty (R1 and R2 in Table 2) associated with the failures of an individual service may be less challenging than integrating a new type of service that becomes available.

Types of requirements that the self-adaptive system must meet: Self-adaptation can be used to achieve different types of system requirements.² Scenarios with a single requirement are typically less challenging for assurance approaches than scenarios

² It is important to distinguish the requirements that the self-adaptive system needs to realize (reliability, performance, etc.) and the requirements for approaches of perpetual assurances.

Table 5. Generic adaptation scenarios for service-based systems

Scenario	Type(s) of uncertainty	Type(s) of requirements	Observable properties	Types of adaptations (examples)
S1	Individual service failure	Reliability	Success or failure of each service invocation (Boolean – true = success, false = failure)	Select equivalent service, invoke idempotent services in parallel, enact alternative service
S2	Variation of quality-of-service property over time (e.g., response time)	Quality of service (e.g. performance)	Variations in the quality of service property (e.g. changes in the response time for each service invocation (ms))	Select equivalent service, invoke idempotent services in parallel, enact alternative service
S3	New alternative service becomes available	Reliability, performance, cost	Available services for each operation (Set of currently available services)	Select new concrete service, enact new service
S4	New type of service becomes available, requirement for new functionality	Add new service	Request to add new functionality (String), available concrete services including services for the new functionality (Set of currently available services)	Adapt workflow, enact adapted workflow, select new concrete service, enact new service

where multiple competing requirements need to be balanced. Different combinations of requirements pose different challenges to perpetual assurances; e.g., monitoring, quantifying, and integrating new evidence (R1, R2, and R5) may be more demanding for requirements that require real-time tracking, while managing overlapping uncertainty resources (R3) and combining new evidence (R6) may be particularly challenging for collecting evidence of interdependent requirements.

Observable attributes/properties: The observations of the system and its execution context (R1 to R7) that are required depend on scenario, and introduce different challenges for adaptation solutions and approaches for perpetual assurances. For example monitoring the failures of a service is less demanding than observing the emergence of a new type of service (which may not have been anticipated upfront) and handling the request for integrating it into the system.

Types of adaptations that are possible: Different scenarios require different types of adaptations. As an example, handling service failures may involve trying equivalent services until an invocation completes successfully, while introducing a new type of

service requires an adaptation of the service workflow. On the other hand, approaches may be available that offer different adaptation strategies; e.g., an alternative strategy for reducing the likelihood of an operation failing may be executing multiple idempotent services in parallel (and using the result returned by the successful execution that completes first).

Different approaches to providing perpetual assurance should be evaluated and compared based on:

1. The benchmark aspects and criteria from Table 4.
2. Their ability to handle the scenarios in Table 5 (or a subset of them).
3. The quality of the adaptation for which they provide assurance, which can be evaluated as described in the existing literature, e.g., [57].

7.2 Tele Assistance System

We now present the case. The concrete application is a Tele Assistance System (TAS) that offers health support to patients using home devices. The example case was presented as an exemplar at SEAMS 2015; for details we refer to [60]. A concrete realisation of the application is available at the community website.³ TAS is based on the example introduced in [6] and later used in [13, 29, 54]. TAS offers a composite service that uses the following services:

- Alarm Service, which provides the operation sendAlarm
- Medical Analysis Service, which provides the operation analyzeData
- Drug Service, which provides the operations changeDoses and changeDrug

Multiple providers offer concrete services for the TAS with different characteristics, e.g. for reliability and cost. Each TAS implementation may use a particular strategy to select concrete services, e.g., based on the minimum response time.

TAS executes the workflow shown in Fig. 1. The system uses the sensors embedded in a wearable medical device to takes periodical measurements of the vital parameters of a patient, and invokes the Medical Analysis Service for their analysis. The analysis result may trigger the invocation of the Drug Service to deliver new medication to the patient or to change his/her dose of medication, or the invocation of the Alarm Service, leading to an ambulance being dispatched to the patient. The Alarm Service can also be invoked directly by the patient by means of a panic button on the wearable device.

7.3 Adaptation Scenarios and Benchmark Criteria

We now provide instances of the generic adaptation scenarios from Table 5.

Setting: The TAS can use one of several concrete services for each abstract service it requires. To select concrete services the workflow uses a service registry with service descriptions that is refreshed from time to time. A service description lists the

³ <https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/tas/>.

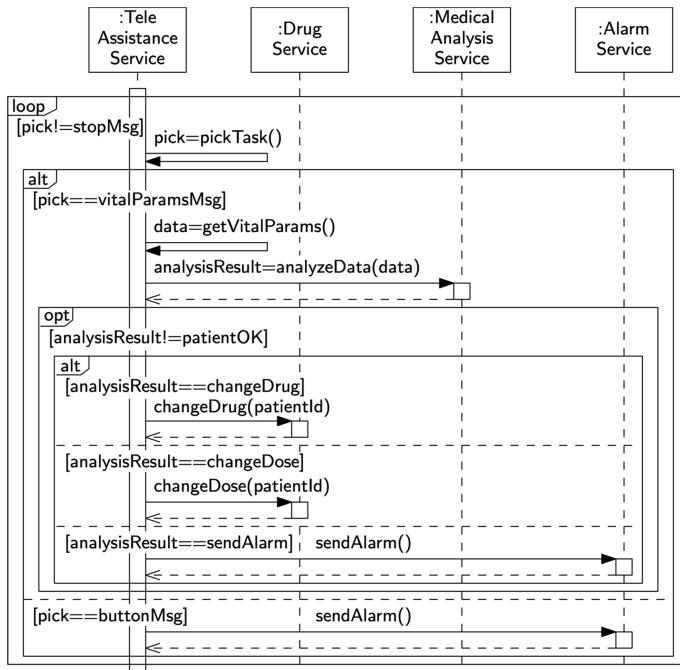


Fig. 1. Workflow of the TAS [60]

maximum failure rate and maximum response time promised by the provider of the concrete service, and the cost per service invocation.

S1 – Individual service failure

Without adaptation: Concrete service i that provides the `sendAlarm` operation is selected such that:

- (i) $fr_i = \min_{1 \leq j \leq n} (fr_j)$, where fr_j is the promised maximum failure rate of the j -th concrete alarm services,
- (ii) $cost_i$ is the minimum cost of the concrete Alarm Service options that satisfy (i).

Uncertainty: The concrete services that provide the Alarm Service fail from time to time (we assume that there are at least two concrete alarm services).

Adaptation requirement: The failure rate of workflow executions that consist of an invocation of the Alarm Service does not exceed a predefined value.

With adaptation: The system observes each successful and failed Alarm Service invocation and dynamically selects the lowest-cost concrete service that meets the requirement.

S2 – Variation of failure rate of services over time

Without adaptation: A pair of concrete services that provide the Medical Analysis Service and Alarm Service is selected such that:

- (i) $fr_1 + fr_2 \leq X$, where fr_1 and fr_2 are the promised maximum failure rates of the two selected services, and X is a pre-specified maximum failure rate of workflow executions comprising an invocation of the Medical Analysis Service followed by an invocation of the Alarm Service.
- (ii) $cost_1 + cost_2$ is minimal cost across all combinations of concrete Medical Analysis Service and Alarm Service options that satisfy (i).

Uncertainty: The failure rates of the concrete services that provide the Alarm Service change significantly over time.

Adaptation requirement: The system requirement of a maximum failure rate of workflow executions comprising an invocation of the Medical Analysis Service followed by an invocation of the Alarm Service lower than or equal to X with minimal cost of service selections is maintained, regardless of the changing failure rates of concrete services over time.

With adaptation: The system observes each successful and failed invocation of a Medical Analysis Service and Alarm Service, uses these observations to estimate the actual failure rates of the concrete services, and dynamically selects the lowest-cost pair of concrete services that meets the requirement.

S3 - New service becomes available

Without adaptation: A concrete service that provides the Alarm Service is selected as in Scenario S1. The registry with service descriptions is periodically refreshed with a period T .

Uncertainty: A new concrete service that provides the Alarm Service becomes available at some time t .

Adaptation requirement: The failure rate of workflow executions that consist of an invocation of the Alarm Service does not exceed a predefined value X .

With adaptation: The system observes each successful and failed invocation of an Alarm Service and dynamically selects the lowest-cost concrete service from the set of concrete alarm services, including, after refreshment of service descriptions, the new concrete service.

S4 – New type of service becomes available

Without adaptation: A concrete service that provides the Alarm Service is selected as in Scenario S1.

Uncertainty: A new type of service called Inform Relatives becomes available at time t ; the Inform Relatives service informs relatives of the patient in case of invocations of the Alarm Service; concrete services of the Inform Relatives service are characterized by a cost.

Adaptation requirement: After a time Δt a concrete service that provides the Inform Relatives service is selected; the lowest cost concrete Inform Relatives service is invoked after every selection of an Alarm Service.

With adaptation: The system discovers new Inform Relatives services, it integrates the corresponding abstract service in the workflow, caches the concrete instances of the new service, and dynamically selects the lowest-cost concrete service from the set of Inform Relatives services.

For each of these scenarios, different benchmark criteria can be applied. For example, for Scenario S2, we may apply the benchmark variability (the capability of an approach to handle variations in requirements) by comparing the percentage of time solutions assure compliance of the requirement that the maximal failure rate of workflow executions is not violated when different distributions of changing failure rates of services are imposed. In Scenario S4, we may benchmark incompleteness by comparing the capability of an approach to provide partial assurances that a new service for contacting relatives is correctly integrated in the workflow. We may apply benchmark performance of approaches for different scenarios by comparing the cost of different approaches to assure compliance with the requirement at runtime, in terms of overhead in time, CPU, and memory.

7.4 Concrete Example Scenario

To conclude this section we illustrate the comparison of two approaches for perpetual assurances for a concrete scenario.

Concrete setup. We use a scenario with an equal number of service instances for each of the three service types used by TAS. An overview of the profiles as declared by third-party service providers for a setup with five instances is shown in Table 6.

Table 6. Third party service profiles for TAS scenario

Service ID	Alarm Service		Medical Analysis Service		Drug Service	
	Failure rate (h^{-1})	Cost (€)	Failure rate (h^{-1})	Cost (€)	Failure rate (h^{-1})	Cost (€)
1	0.11	4.0	0.12	4.0	0.01	5.0
2	0.04	12.0	0.07	14.0	0.03	3.0
3	0.18	2.0	0.18	2.0	0.05	2.0
4	0.08	3.0	0.10	6.0	0.07	1.0
5	0.14	5.0	0.15	3.0	0.02	4.0

Table 7 shows the distributions of requests in the TAS scenario.

Table 7. Distribution of requests in the TAS scenario (as a fraction of all requests)

Distribution user requests		Distribution requests after analysis	
Checking vital signs	Emergency request	Drug service	Alarm service
0.75	0.25	0.66	0.34

The service profile parameters and the distribution of requests are subject to uncertainty. Concretely, we added uncertainty to the failure rates of services and the distribution of requests based on a normal distribution.

Requirements. The concrete requirements that should be fulfilled are:

- R1. The failure rate of invocations of TAS per hour should be below 0.02.
- R2. The average cost per invocation of TAS should be below 8.0 ¢.

Hence, the scenario is a concrete instance of abstract scenario S1 (individual service failure) combined with scenario S2 (variation of failure rates of services over time).

Approaches. We apply and compare two approaches for perpetual assurance: runtime quantitative verification (RQV) [13] and runtime statistical model checking (RSMC) [40]. RQV applies model-based evaluation of qualities using probabilistic verification techniques. Concretely, the approach models the TAS service system as a discrete-time Markov chain and uses the PRISM model checker [44] at runtime to assure that service configurations are selected that comply to the requirements. The Markov model is parameterized by the configurable parameters of the service-based system (i.e., service profile parameters and distributions of requests to TAS) that are updated at runtime. During analysis the configurations that satisfy the quality requirements for the system are identified. The planner uses the analysis results to create a plan for adapting the configuration as needed. RQV assures that selected configurations comply with the requirements. RSMC on the other hand applies model-based evaluation of qualities using statistical verification techniques. The approach models the TAS systems with stochastic timed automata and uses the Uppaal-SMC model checker [24] at runtime to select service configurations that comply to the requirements. Similar to the Markov model used with RQV, the stochastic timed automata model is parameterized with service profile parameters and changing distributions of requests to TAS that are updated at runtime. RSMC checks the required properties of the model based on a sample set of simulations. RSMC uses statistical techniques to decide whether the system satisfies the property with some degree of confidence. During analysis the configurations that satisfy the quality requirements with a defined degree of confidence are identified. The planner uses these configurations to create a plan for adapting the configuration as needed. Thus, in contrast to exhaustive approaches, such as RQV, RSMC does not provide 100% guarantees, but an estimation, which is bound to a confidence interval. On the other hand, by setting the verification parameters, SMCR allows to tradeoff between the accuracy and confidence of the guarantees it provides with the system resources it requires.

The left-hand side of Fig. 2 shows the failure rates obtained when RSMC was used for the concrete setup shown in Tables 6 and 7. The boxplots show averages of 20 runs over 5 h (with approximately 1000 invocations per hour). We used four different settings of verification parameters E and A (RSMC Strategy). The value of E defines the approximation interval, i.e. $[p - E, p + E]$ with p the probability that invocations of TAS fail, and $1 - A$ is the confidence level for the result. The figure shows that the setting with $A = 0.05$ and $E = 0.05$ guarantees no violations of the failure rate with a confidence level of 95%. The results with more relaxed parameter settings provide a less accurate result with lower confidence levels, but the mean values for all considered

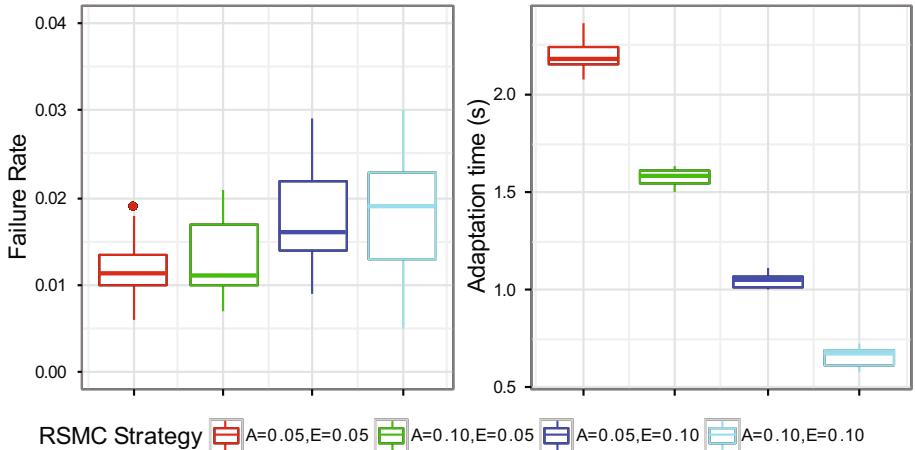


Fig. 2. Failure rates (left) and computation time (right) with RSMC

strategies remain under the requirement constraint of 2%. The right-hand side of the figure shows that the time required to compute the verification results decreases when the accuracy and confidence of the results is lowered. The approach enables balancing the tradeoff between computation time and accuracy/confidence of runtime model checking.

We compared the adaptation time with RQV and RSMC for settings with an increasing number of service instances for TAS. Figure 3 shows the results of the experiments. The graphs confirm that RSMC is more efficient in required computation time and scales better. However, RQV can provide strict guarantees for the requirements at hand, while the efficiency of RSMC comes at an expense since fewer simulations can lead to results with less accuracy and confidence.

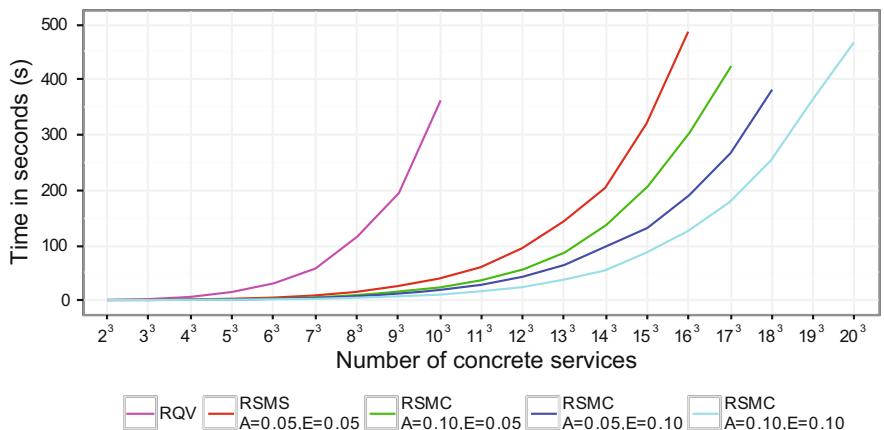


Fig. 3. Adaptation time for RQV versus RSMC

We summarize the comparison for RQV and RSMC for a set of applicable benchmark criteria defined in Table 8.

Table 8. Comparison benchmark criteria RQV versus RSMC

Benchmark criteria			
Criteria	Description	RQV	RSMC
Inaccuracy & Incompleteness	Capability of an approach to handle inaccuracy and incompleteness of models of the system and context	Yes through the use of a parameterized Markov model	Yes by using a parameterized stochastic timed automata with configurable verification parameters
Conflicting criteria	Capability of an approach to balance the tradeoffs between utility (correctness) and cost (verification time, memory)	Exact approach; no support to balance correctness versus verification time	Tradeoff between accuracy and confidence and required verification time and memory
Combined approaches	Capability of an approach to provide evidence based on combining historical data with predictive models	Both approaches provide evidence based on the prediction of expected behavior based on past observed data	
Timeliness	The time an approach requires to achieve the required evidence	Exhaustive approach; expected to be limited to smaller modes than RSMC	Configurable based on setting of verification parameters

8 Conclusions

Assuring requirements compliance of self-adaptive systems that have to operate under uncertainty calls for an enduring process where evidence is collected over the whole lifetime of the system. We coined this process as perpetual assurances for self-adaptive systems. We summarize the key challenges to realize perpetual assurances.

First, we need a better understanding of the nature of uncertainty for software systems and how this poses requirements for providing perpetual assurances. This paper provides an initial list of requirements for perpetual assurances based on a proposed classification of uncertainties. Additional research is required to test the validity and coverage of this set of requirements.

Second, we need a deeper understanding of how we can handle uncertainty in self-adaptive systems, and in particular, how can we monitor and quantify uncertainty.

Currently, there is a growing understanding of how to handle uncertainty regarding parameters of the system, its goals, and the environment. However, how to handle uncertainty regarding parts of the system and the environment that may not be completely known upfront, or handling uncertainty regarding new goals remains to a large extent an open problem.

Third, deriving, integrating and combining new evidence pose additional hard challenges. A variety of techniques for obtaining evidence for requirements compliance of software systems exist. However, most of these techniques have been conceived for offline use. Perpetual assurance requires continuously deriving, integrating and combining new evidence, while the system is operating. We have presented decomposition and model-based mechanisms than can potentially pave the paths to go forward. However, making these mechanisms effective is particularly challenging and requires a radical revision of many of the existing techniques.

Last but not least, to drive research on assurances for self-adaptive systems forward, we need good exemplars. Exemplars enable comparison of different solution, pinpoint the critical challenges, and demonstrate the effectiveness of the different mechanisms adopted by the self-adaptive solutions. The case used in this paper that is further explained in [60] provides one exemplar in the domain of service-based systems.

References

1. Andersson, J., Baresi, L., Bencomo, N., de Lemos, R., Gorla, A., Inverardi, P., Vogel, T.: Software engineering processes for self-adaptive systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-adaptive Systems II. LNCS, vol. 7475, pp. 51–75. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_3
2. Amin, A., Colman, A., Grunske, L.: An approach to forecasting QoS attributes of web services based on ARIMA and GARCH models. In: International Conference on Web Services, ICWS 2012, pp. 74–81 (2012)
3. Amin, A., Grunske, L., Colman, A.: An automated approach to forecasting QoS attributes based on linear and non-linear time series modeling. In: ASE 2012, pp. 130–139 (2012)
4. Ardagna, D., Zhang, L. (eds.): Run-time Models for Self-managing Systems and Applications. Springer, Cham (2010). <https://doi.org/10.1007/978-3-0346-0433-8>
5. Autili, M., Cortellessa, V., Di Ruscio, D., Inverardi, P., Pelliccione, P., Tivoli, M.: Integration architecture synthesis for taming uncertainty in the digital space. In: Calinescu, R., Garlan, D. (eds.) Monterey Workshop 2012. LNCS, vol. 7539, pp. 118–131. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34059-8_6
6. Baresi, L., Bianculli, D., Ghezzi, C., Guinea, S., Spoletini, P.: Validation of web service compositions. IET Software **1**(6), 219–232 (2007)
7. Baresi, L., Ghezzi, C.: The disappearing boundary between development-time and runtime. In: FSE/SDP Workshop on Future of Software Engineering Research. ACM (2010)
8. Bencomo, N.: QuantUn: quantification of uncertainty for the reassessment of requirements. In: Next! track at the 23rd International Requirements Engineering Conference (2015)
9. Bencomo, N., France, R., Cheng, B.H.C., Aßmann, U. (eds.): Models@run.time. LNCS, vol. 8378. Springer, Cham (2014). <https://doi.org/10.1007/978-3-319-08915-7>

10. Beugnard, A., Jezequel, J., Plouzeau, N., Watkins, D.: Making components contract aware. *IEEE Computer* **32**(7), 38–45 (1999)
11. Calinescu, R., Kwiatkowska, M.: Using quantitative analysis to implement autonomic IT systems. In: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), pp. 100–110. IEEE Computer Society, Washington, DC, USA (2009)
12. Calinescu, R., Ghezzi, C., Kwiatkowska, M., Mirandola, R.: Self-adaptive software needs quantitative verification at runtime. *Commun. ACM* **55**, 9 (2012)
13. Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G.: Dynamic QoS management and optimization in service-based systems. *IEEE Trans. Software Eng.* **37**(3), 387–409 (2011)
14. Calinescu, R., Johnson, K., Rafiq, Y.: Developing self-verifying service-based systems. In: IEEE/ACM 28th International Conference on Automated Software Engineering (2013)
15. Calinescu, R., Johnson, K., Rafiq, Y.: Using observation ageing to improve markovian model learning in QoS engineering. In: 2nd ACM/SPEC International Conference on Performance Engineering. ACM, New York (2011)
16. Calinescu, R., Kikuchi, S., Johnson, K.: Compositional reverification of probabilistic safety properties for large-scale complex IT systems. In: Calinescu, R., Garlan, D. (eds.) Monterey Workshop 2012. LNCS, vol. 7539, pp. 303–329. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34059-8_16
17. Calinescu, R., Rafiq, Y., Johnson, K., Bakır, M.E.: Adaptive model learning for continual verification of non-functional properties. In: 5th ACM/SPEC International Conference on Performance Engineering, pp. 87–98. ACM, New York (2014)
18. Cámara, J., de Lemos, R., Laranjeiro, N., Ventura, R., Vieira, M.: Testing the robustness of controllers for self-adaptive systems. *J. Braz. Comput. Soc.* **20**, 1 (2014)
19. Camara, J., Moreno, G., Garlan, D., Schmerl, B.: Analyzing latency-aware self-adaptation using stochastic games and simulations. *ACM Trans. Auton. Adapt. Syst.* **10**(4), 23:1–23:28 (2016)
20. Cardellini, V., Casalicchio, E., Grassi, V., Iannucci, S., Lo, P.F., Mirandola, R.: MOSES: a framework for QoS driven runtime adaptation of service-oriented systems. *IEEE Trans. Softw. Eng.* **38**(5), 1138–1159 (2012)
21. Cavallo, B., Di Penta, M., Canfora, G.: An empirical comparison of methods to support QoS-aware service selection. In: PESOS 2010, pp. 64–70 (2010)
22. Cheng, B.H.C.: Software engineering for self-adaptive systems: a research roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_1
23. Cheng, B.H.C., et al.: Using models at runtime to address assurance for self-adaptive systems. In: Bencomo, N., France, R., Cheng, B.H.C., Aßmann, U. (eds.) Models@run.time. LNCS, vol. 8378, pp. 101–136. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08915-7_4
24. David, A., Larsen, K., Legay, A., Mikulaonis, M., Poulsen, D.: Uppaal SMC tutorial. *Int. J. Softw. Tools Technol. Transfer* **17**(4), 397–415 (2015)
25. de la Iglesia, D.G., Weyns, D.: Guaranteeing robustness in a mobile learning application using formally verified MAPE loops. In: Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2013 (2013)
26. de la Iglesia, D.G., Weyns, D.: MAPE-K formal templates to rigorously design behaviors for self-adaptive systems. *ACM Trans. Auton. Adaptive Syst.* **10**(3), 15 (2015)

27. de Lemos, R., et al.: Software engineering for self-adaptive systems: a second research roadmap. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) *Software Engineering for Self-adaptive Systems II*. LNCS, vol. 7475, pp. 1–32. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_1
28. Epifani, I., Ghezzi, C., Tamburrelli, G.: Change-point detection for black-box services. In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2010)
29. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by runtime parameter adaptation. In: *Proceedings of the 31st International Conference on Software Engineering* IEEE Computer Society (2009). 2009.5070513
30. Esfahani, N., Malek, S.: Uncertainty in self-adaptive software systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) *Software Engineering for Self-adaptive Systems II*. LNCS, vol. 7475, pp. 214–238. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_9
31. Filieri, A., Ghezzi, C., Tamburrelli, G.: Runtime efficient probabilistic model checking. In: *International Conference on Software Engineering, ICSE 2011* (2011)
32. Filieri, A., Grunske, L., Leva, A.: Lightweight adaptive filtering for efficient learning and updating of probabilistic models. In: *International Conference on Software Engineering, ICSE 2015*, pp. 200–211 (2015)
33. Fredericks, E.M., Ramirez, A.J., Cheng, B.H.C.: Towards runtime testing of dynamic adaptive systems. In: *8th International Symposium on Software Engineering for Adaptive and Self-managing Systems* (2013)
34. Funtowicz, S., Ravetz, J.: *Uncertainty and Quality in Science for Policy*. Springer, Dordrecht (1990). <https://doi.org/10.1007/978-94-009-0621-1>
35. Garlan, D.: Software engineering in an uncertain world. In: *Future of Software Engineering Research Workshop, FoSER*, New York, NY, USA (2010)
36. Giese, H., Bencomo, N., Pasquale, L., Ramirez, Andres J., Inverardi, P., Wätzoldt, S., Clarke, S.: Living with uncertainty in the age of runtime models. In: Bencomo, N., France, R., Cheng, B.H.C., Abmann, U. (eds.) *Models@run.time*. LNCS, vol. 8378, pp. 47–100. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08915-7_3
37. Grunske, L., Zhang, P.: Monitoring probabilistic properties. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 183–192. ACM (2009)
38. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: *PARAM*: a model checker for parametric Markov Models. In: Touili, T., Cook, B., Jackson, P. (eds.) *CAV 2010*. LNCS, vol. 6174, pp. 660–664. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_56
39. Iftikhar, U., Weyns, D.: ActivFORMS: active formal models for self-adaptation. In: *Software Engineering for Adaptive and Self-managing Systems, SEAMS 2014* (2014)
40. Iftikhar, U., Weyns, D.: A case for runtime statistical model checking for self-adaptive systems, Technical report Katholieke Universiteit Leuven CW 693 (2016). <http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW693.abs.html>
41. Inverardi, P., Mori, P.: Model checking requirements at runtime in adaptive systems. In: *Workshop on Assurances for Self-adaptive Systems* (2011)
42. Inverardi, P.: Software of the future is the future of software? In: Montanari, U., Sannella, D., Bruni, R. (eds.) *TGC 2006*. LNCS, vol. 4661, pp. 69–85. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75336-0_5
43. Johnson, K., Calinescu, R., Kikuchi, S.: An incremental verification framework for component-based software systems. In: *CBSE 2013*, pp. 33–42 (2013)

44. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
45. Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Assume-guarantee verification for probabilistic systems. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 23–37. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_3
46. Kwiatkowska, M., Parker, D., Qu, H.: Incremental quantitative verification for Markov decision processes. In: Proceedings 2011 IEEE/IFIP International Conference Dependable Systems and Networks (2011)
47. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: an overview. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 122–135. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_11
48. Mahdavi-Hezavehi, S., Avgeriou, P., Weyns, D.: A classification of current architecture-based approaches tackling uncertainty in self-adaptive systems with multiple requirements. In: Managing Trade-offs in Adaptable Software Architectures. Elsevier (2016)
49. Mula, J., Poler, R., Garcia-Sabater, J., Lario, F.: Models for production planning under uncertainty: a review. *IJPE* **103**(1), 271–285 (2006)
50. Pavese, E., Braberman, V., Uchitel, S.: Probabilistic environments in the quantitative analysis of (non-probabilistic) behaviour models. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pp. 335–344. ACM, New York
51. Perez-Palacin, D., Mirandola, R.: Uncertainties in the modeling of self-adaptive systems: a taxonomy and an example of availability evaluation. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering. ACM, New York (2014)
52. Ramirez, A., Jensen, A., Cheng, B.: A taxonomy of uncertainty for dynamically adaptive systems. In: Software Engineering for Adaptive and Self-Managing Systems (2012)
53. Refsgaard, J.C., van der Sluijs, J.P., Højberg, A.L., Vanrolleghem, P.A.: Uncertainty in the environmental modeling process - a framework and guidance. *Environ. Model. Softw.* **22**(11), 1543–1556 (2007)
54. Shevtsov, S., Iftikhar, U., Weyns, D., SimCA vs ActivFORMS: comparing control- and architecture-based adaptation on the TAS exemplar. In: Control Theory and Software Engineering, CTSE 2015 (2015)
55. Sykes, D., Corapi, D., Magee, J., Kramer, J., Russo, A., Inoue, K.: Learning revised models for planning in adaptive systems. In: Proceedings of the 2013 International Conference on Software Engineering (ICSE 2013), pp. 63–71. IEEE Press, Piscataway (2013)
56. Uttamchandani, S., Yin, L., Alvarez, G., Palmer, J., Agha, G.: CHAMELEON: a self-evolving, fully-adaptive resource arbitrator for storage systems. In: USENIX Technical Conference 2005 (2005)
57. Villegas, N., Müller, H., Tamura, G., Duchien, L., Casallas, R.: A framework for evaluating quality-driven self-adaptive software systems. In: 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 80–89 (2011)
58. Vogel, T., Giese, H.: Model-driven engineering of self-adaptive software with EUREMA. *ACM Trans. Auton. Adapt. Syst.* **8**(4), Article 18 (2014)
59. Walker, W., Harremos, P., Romans, J., van der Sluus, J., van Asselt, M., Janssen, P., Krauss, M.: Defining uncertainty. a conceptual basis for uncertainty management in model-based decision support. *Integr. Assess.* **4**(1), 5–17 (2003)
60. Weyns, D., Calinescu, R.: Tele assistance: a self-adaptive service-based system exemplar. In: Software Engineering for Adaptive and Self-Managing Systems (2015)
61. Weyns, D., Iftikhar, U.: Model-based simulation at runtime for self-adaptive systems. In: Models at Runtime (2016)

62. Weyns, D.: Software engineering of self-adaptive systems: an organised tour and future challenges. In: Dick Taylor, R., Kang, K., Cha, S. (eds.) *Handbook of Software Engineering*, Springer (2018, forthcoming). <https://people.cs.kuleuven.be/danny.weyns/papers/2017HSE.pdf>
63. Weyns, D., Iftikhar, M.U., de la Iglesia, D.G., Ahmad, T.: A survey of formal methods in self-adaptive systems. In: Fifth International C* Conference on Computer Science and Software Engineering, C3S2E 2012, pp. 67–79. ACM, New York (2012)
64. Weyns, D., Malek, S., Andersson, J.: FORMS: unifying reference model for formal specification of distributed self-adaptive systems. *ACM Trans. Auton. Adaptive Syst. TAAS* **7**(1), 8 (2012)
65. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H., Bruel, J.M.: Relax: incorporating uncertainty into the specification of self-adaptive systems. In: Requirements Engineering Conference (2009)
66. Ye, C., Cheung, S.C., Chan, W.K.: Process evolution with atomicity consistency. In: *Software Engineering of Adaptive and Self-Managing Systems, SEAMS 2007* (2007)
67. Younes, H.L.S., Simmons, R.G.: Statistical probabilistic model checking with a focus on time-bounded properties. *Inform. Comput.* **204**(9), 1368–1409 (2006)
68. Younes, H.L.S., Kwiatkowska, M.Z., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking. *STTT* **8**(3), 216–228 (2006)
69. Zhang, J., Cheng, B.: Using temporal logic to specify adaptive program semantics. *J. Syst. Softw.* **79**, 1361–1369 (2006)
70. Zhang, J., Cheng, B.: Model-based development of dynamically adaptive software. In: 28th International Conference on Software Engineering (2006)

Challenges in Composing and Decomposing Assurances for Self-Adaptive Systems

Bradley Schmerl¹(✉), Jesper Andersson², Thomas Vogel³, Myra B. Cohen⁴, Cecilia M. F. Rubira⁵, Yuriy Brun⁶, Alessandra Gorla⁷, Franco Zambonelli⁸, and Luciano Baresi⁹

¹ Carnegie Mellon University, Pittsburgh, PA, USA

schmerl@cs.cmu.edu

² Linnaeus University, Växjö, Sweden

jesper.andersson@lnu.se

³ Hasso Plattner Institute, University of Potsdam, Potsdam, Germany

thomas.vogel@hpi.de

⁴ University of Nebraska, Lincoln, NE, USA

myra@cse.unl.edu

⁵ University of Campinas, Campinas, SP, Brazil

cmrubira@ic.unicamp.br

⁶ University of Massachusetts, Amherst, MA, USA

brun@cs.umass.edu

⁷ IMDEA Software Institute, Madrid, Spain

alessandra.gorla@imdea.org

⁸ University of Modena and Reggio Emilia, Modena, Italy

franco.zambonelli@unimore.it

⁹ Politecnico di Milano, Milan, Italy

luciano.baresi@polimi.it

Abstract. Self-adaptive software systems adapt to changes in the environment, in the system itself, in their requirements, or in their business objectives. Typically, these systems attempt to maintain system goals at run time and often provide assurance that they will meet their goals under dynamic and uncertain circumstances. While significant research has focused on ways to engineer self-adaptive capabilities into both new and legacy software systems, less work has been conducted on how to assure that self-adaptation maintains system goals. For traditional, especially safety-critical software systems, assurance techniques decompose assurances into sub-goals and evidence that can be provided by parts of the system. Existing approaches also exist for composing assurances, in terms of composing multiple goals and composing assurances in systems of systems. While some of these techniques may be applied to self-adaptive systems, we argue that several significant challenges remain in applying them to self-adaptive systems in this chapter. We discuss how existing assurance techniques can be applied to composing and decomposing assurances for self-adaptive systems, highlight the challenges in applying them, summarize existing research to address some of these challenges, and identify gaps and opportunities to be addressed by future research.

1 Introduction

Modern software systems typically have to operate in complex and diverse environments and conditions. For example, business and cloud-based systems must cater for a wide range of load and customer profiles, and systems that manage physical elements must deal with uncertainties in the physical world. Self-adaptive systems form a category of software that changes, reconfigures, or fixes itself as it is running. Much research has been conducted into different methods for constructing self-adaptive systems, for example by integrating control loops to manage systems or by using self-organizing or bio-inspired principles [42]. Self-adaptive systems often attempt to maintain or achieve system goals in the face of uncertainty, and are usually constructed to provide some confidence that a system at run time will continue to operate appropriately, even in changing and uncertain circumstances.

While various methods for constructing self-adaptive systems have proven successful in a number of domains, *assuring* the self-adaptive aspects of these systems remains a challenge. Assuring self-adaptive systems requires run-time validation and verification (V&V) activities [45]. This is mainly because the combination of self-adaptive configurations and the environments that they encounter leads to a state explosion that makes static V&V challenging. One way to address this challenge is to apply techniques for decomposing and composing assurances. For safety-critical systems there is a large body of work on constructing safety cases, or more generally assurance cases, that construct assurance arguments about these kinds of systems. Reasoning about assurances in safety-critical systems may shed some light on how to provide these assurances for self-adaptive systems. Typically, assurances involve decomposing top level goals into argumentation structures that involve sub-goals, strategies for achieving the goals, and defining evidence that can be collected to show that the goals are achieved. Top level goals can also be composed together to provide assurances about a system with multiple goals, to reuse some assurances for goals in similar systems, or to provide assurances in systems of systems.

In this chapter we discuss the challenges related to decomposing and composing assurances in self-adaptive systems. In Sect. 2 we give some background on assurances, focusing on assurance cases as a framework for guiding decomposition and composition of assurances. We also introduce an example that will be used to illustrate the challenges. In Sect. 3, we survey existing self-adaptation assurance research that has either discussed how to compose assurances, or could be used to help build an assurance argument. Section 4 identifies a set of challenges associated with composing and decomposing assurances. In Sect. 5 we discuss some emerging research in assurance cases that should be followed to help with composition and decomposition and outline the challenges that arise when applying assurance cases to the context of self-adaptation. In Sect. 6 we provide some concluding remarks.

2 Preliminaries

In this section, we briefly introduce self-adaptive systems as we conceive them in the scope of this chapter. Then, we discuss techniques for software assurance in safety-critical systems, and describe an illustrative example that we use throughout the chapter.

2.1 Self-Adaptive Systems

Current and emerging software systems are increasingly complex and distributed, and are called to operate in open-ended and unpredictable operational environments. On one hand, such uncertainty challenges the capabilities of a system to maintain its business goals if the configuration is to remain static. On the other hand, changing environments or the system itself may also modify the goals and requirements for which the system was originally structured and configured.

To tackle the above situation, human intervention has historically been required. However, human intervention is generally impossible due to the inherent decentralized nature of modern systems, or simply infeasible due to economic or temporal reasons. Accordingly software systems have to become *self-adaptive* in their behaviour, that is, capable of dynamically adapting their configuration and/or structure in an autonomous way without human supervision, in order to respond to changing situations without malfunctioning or degrading quality of service unacceptably [16, 42].

In modern software systems, self-adaptation can take place both via mechanisms integrated in individual components as well as in groups/collectives of components (e.g., [8–12]), and that have the goal of modifying something in the behaviour of a component or a collective (e.g., [6, 7]).

The study of both individual and collective adaptation mechanisms has a long history. Individual adaptation is a very important thread of research since the early years of intelligent agents [35] and reflective computing [48], and several architectures and mechanisms to enable adaptation have been proposed so far, including the recent IBM autonomic computing approach [38]. All proposals for self-adaptation at the level of individual components rely on the integration, within each component, of a closed control loop. In the control loop, a specific control component (e.g., the “autonomic manager” in the autonomic computing approach, or the “meta-component” in reflective approaches) monitors and analyses the current operational and environmental condition of the component, and plans and executes appropriate adaptation actions as needed. The predominant pattern for self-adaptation that has emerged for structuring an autonomic manager is MAPE-K [38], where each of the activities that need to occur as part of adaptation are Monitoring – or sensing – the system and the environment, Analysing the system to determine whether the current state of the system requires adaptation, Planning, which determines what adaptations to perform, and Executing to effect changes in the system. All of this is coordinated through Knowledge.

For collective adaptation, the simplest approach is to integrate a single controller in charge of managing a whole collective with a single control loop, but this approach has challenges when scaling to realistic systems. For this reason, a variety of patterns for coordinating multiple controllers and control loops has been investigated [41, 50].

For both individual and collective adaptation uniform models and tools supporting the design and development of self-adaptive systems are still missing. Furthermore, there are few methods for assuring that self-adaptive systems adapt correctly, with respect to performing as designed to achieve their intended goals, doing so in a safe and consistent manner, and ensuring that adaptations result in legal systems respecting their design and business constraints.

2.2 Assurance Cases

Self-adaptation can be decomposed into a number of activities the use of which can span from design to run time. It is therefore not possible to provide a single assurance mechanism to provide guarantees about self-adaptation. In fact, as we shall see, there exists a collection of assurance techniques that can be used during these various activities. These techniques need to be applied and structured in a principled way in order to provide assurances. We need, therefore, to carefully consider how assurances should be decomposed into these assurance activities to ensure that the activities do in fact help assure overall goals, and to ensure that we are only doing assurance activities that help meet our goals. Furthermore, because self-adaptive systems are increasingly being composed, we need methods and approaches for composing the systems' associated assurances to assure global properties about the collective adaptive system.

To do both of these things, we can look at how assurances are handled in safety-critical systems. In this section, we discuss some solutions for software assurance and propose that assurance cases could be a good starting point for decomposing and composing assurances for self-adaptive systems. In the area of safety critical systems, there has been considerable research in software assurances. An assurance can be defined as a justified measure of confidence that a system will function as intended in its environment of use.

Assuring that a system satisfies some quality and functional goals requires the construction and evaluation of a reasoning approach based on claims, arguments, evidence, and expertise. For example, a *safety case* presents a structured demonstration that a system is acceptably safe in a given context. In other words, it is a comprehensive presentation of evidence linked by argument to a claim. For example, if we are trying to assure a claim *Claim1*, then an assurance case might decompose this claim into two subclaims, *Claim2* and *Claim3* that are easier to show, with some argument that says that if *Claim2* and *Claim3* are true, then *Claim1* is true. We could then provide some evidence that shows that each of *Claim2* and *Claim3* are true. Structuring evidence in such a way means that an expert can make a judgement that the argument makes sense and thus, if that evidence is provided, have confidence that the system is acceptably safe.

Assurance cases are a generalization of safety cases to construct arguments that are about more than just safety.

While this rationale can be presented textually in documentation, it has proven useful to use graphical notations that help define and present the argumentation structure for assurance cases. The structure of an assurance case can be graphically represented using, for instance, the Goal Structuring Notation (GSN) [37] or Claims-Argument-Evidence (CAE) [5]. GSN is a well-accepted graphical notation to show how claims (or goals) can be broken down into sub-claims, and eventually supported by evidence, making clear the argumentation strategies adopted, the rationale for the approach (assumptions, justifications), and the context in which claims are stated. In general, arguments are structured hierarchically: claim, argument, sub-claims, sub-arguments, evidence. It is essential that assurance cases are presented in a clear structure, and GSN can capture the elements most critical for arguing a case (claims, evidence, argument strategy, assumptions, relation of claims to sub-claims and evidence) to build a convincing case.

2.3 Illustrative Example

Throughout this paper we will use a simple example to illustrate how assurances could be decomposed and composed for self-adaptive systems, and some of the challenges in doing so. The example that we will use is Znn, a typical web system serving news articles and related images, that is implemented as a three-tiered web service using a standard LAMP stack (Linux, Apache, MySQL, PHP). Figure 1 shows the Znn architecture with one dispatcher, which shares the load evenly between two web servers, and one database, which stores images for served articles. While Znn itself is not self-adaptive, it provides APIs that allow a control loop to be added onto it to manage quality of service goals. Examples of a quality goal include keeping the response time below two seconds, which is related to how many servers can be used by the dispatcher and the detail of the content that each server produces (e.g., as reported in [18]). Another related goal might be to keep the operational costs below a certain threshold.

To support control loops on top of it, Znn provides a number of APIs for probing the state of the system and effecting changes. For example, it is possible to affect the configuration of Znn by changing the number of web servers or modifying the detail of the content served. Both of these simple changes can affect response time, which is information that can be retrieved from Znn using probes.

Thus, self-adaptation can be added to Znn by integrating a control loop that takes inputs from Znn probes and effects changes on Znn, via the API. For example, Rainbow [28] takes the probe inputs, abstracts them to values in the architectural model of the system, and then conducts an analysis of this architecture to determine if something is wrong (e.g., the response time is too high). If corrective action should be taken, Rainbow balances various business quality concerns in order to decide the best effects to make in the system [17]. For example, it will trade-off increasing the number of servers (thereby increasing

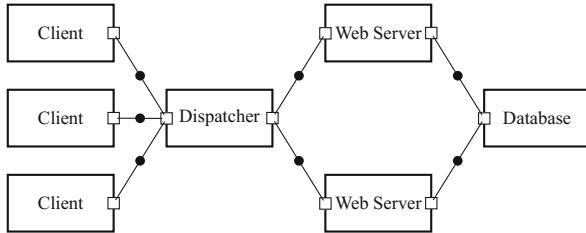


Fig. 1. Architecture of the Znn web system.

costs), decreasing content detail, with the effect these will have on response time, and choose an option that has the highest overall utility.

Znn is an interesting example for decomposing assurances because we would like to assure that the response time is below the threshold. For example, we want to be able to answer questions such as how does one construct this argument, and what forms of evidence can be provided by a self-adaptive system? We can consider two aspects of composition of assurances in Znn. First, if there is a self-adaptive system that is trying to assure response time goals, and we want to combine this with an assurance that costs do not go above a certain amount, we want to be able to reason about how the assurances can be defined, whether there are any conflicts and how they are resolved, and the kinds of evidence and strategies that can be used to reason about the assurance. Second, we are interested in similar questions if two self-adaptive systems are being composed (e.g., Znn, for which performance and cost are important, and another system that uses the same infrastructure but has an additional goal of security).

Assurance Cases for Znn Self-Adaptation. As discussed in Sect. 2.2, assurance cases could be used to organize assurances for self-adaptive systems and to identify evidence that can be provided for those assurances. Among others, such evidence can be based on observations, testing, simulation, and the process used to construct the system. The argument around an assurance case represents a high-level explanation of how evidence combines to show that the goals (or claims) will be met. The evidence and arguments are usually structured as a tree, with high-level goals being decomposed into increasingly fine-grained sub-goals that are eventually supported by evidence.

As mentioned in Sect. 2.2, one common way to document assurance cases is to use the Goal Structuring Notation (GSN) [30] to structure the assurance case as a tree. GSN has nodes for *claims* (or *goals*) that need to be shown and form part of the argument. These can be decomposed into sub-claims or strategies. *Strategies* describe how the claim is to be shown for the assurance case, and then *evidence* or *solutions* are activities or evidence that is used to support the claim. Associated with each claim or strategy is a *context*, which states the assumptions under which the claim is made or in which the strategy is valid. If it is necessary to state the assumptions that a strategy relies on to be valid, or

to justify a strategy, these can be documented via *Assumptions* or *Justifications* in the assurance case. Finally, goals or strategies that have not been decomposed can be denoted by placing a diamond underneath them in the graphical notation. The graphical legend for these items in GSN are denoted in Fig. 2.

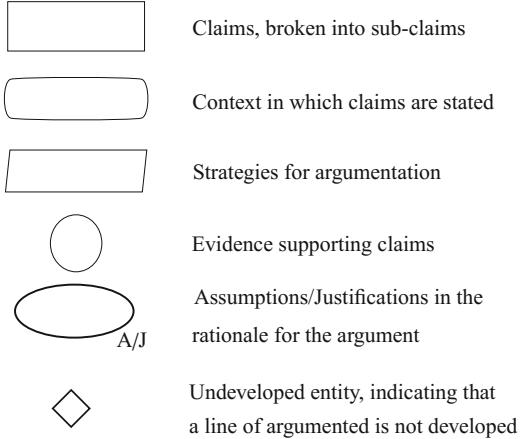


Fig. 2. Goal Structuring Notation legend.

Consider a high-level goal for Znn that it will be able to reply to all requests within 2 s. An example of an assurance case for this goal is shown in Fig. 3. An engineer may choose a strategy of designing two versions of Znn, one for normal operation (G2) and one for high-load operation (G3), and a method for adapting the system by switching between these versions when the assumptions change (i.e., the user load changes). The sub-goals G2 and G3 would then show that the response time goals are met in each of these versions under the different load contexts. To obtain evidence for each of these sub-goals, architectural performance analysis (e.g., based on queuing theory) for that version and validation processes throughout development such as component-level testing might be used. In this context, assumptions concerning individual components that are made in the architectural analysis can be supported by evidence from component-level tests. Evidence would then be the results of the analysis and tests, which give one the confidence that each sub-goal is met in its context. In this way, an assurance is decomposed into sub-goals and evidence while the strategies form the arguments for the assurance case.

The assurances cases for G2 and G3 would proceed as normal for the static design and assurance of the various Znn modes/versions. We will not discuss these further here. Instead, we are concerned with composing and decomposing assurances that relate to the *self-adaptive* part of the system. Such an example happens in Znn when the user load forces a change in modes (e.g., the load changes from 3000 requests per second to 7000 requests per second). In this

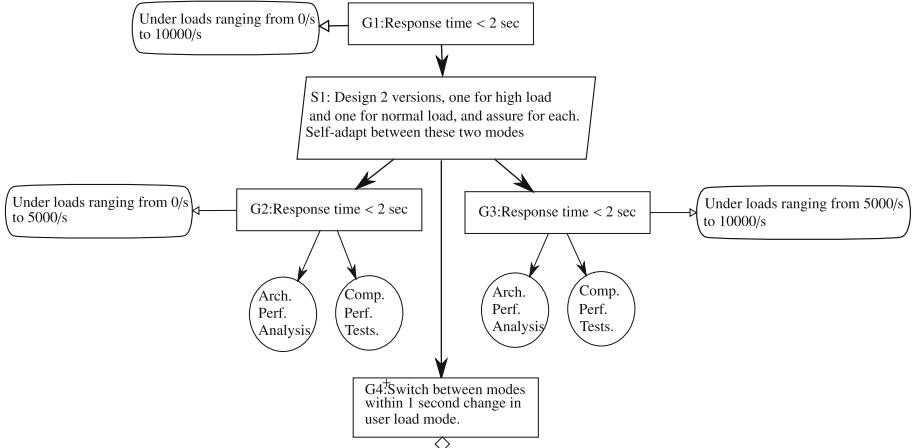


Fig. 3. An example Goal Structuring Notation diagram for Znn.

case, we want to provide the assurance that the mode switch happens within 1 s (G4). We will elaborate the assurance case of G4 to illustrate both, how we might use assurance cases to guide decomposition and composition, and to highlight some of the challenges associated with each of these.

3 Assurance Decomposition and Composition in Self-Adaptive Systems

The focus of much of the research in self-adaptive systems to date has been to engineer systems that can maintain stated goals, even in the presence of uncertain and changing environments. There is existing research in assurances for self-adaptive systems that either addresses how to compose assurances, or can be used as part of an argument in assurance cases. The purpose of this section is to summarize some of this research, and to illustrate how it might apply to the problem of decomposition and composition specifically.

As argued above, it is not possible to provide a single, monolithic assurance for the goals of a self-adaptive system. Assurance cases can provide a way to organize existing work on assurances into self-adaptive system. We can organize this existing work in the following areas:

Evidence types and sub-goals for use in assurance case decomposition.

Each of the classic activities of self-adaptation - monitoring, analysis, planning, and execution - have existing techniques that help to provide evidence for goals that can be used in assurance cases. These approaches could be used in assurance case decomposition.

Assurance composition based on the MAPE-K loop. Once assurances have been decomposed, we need ways to recompose the assurances. Many

of these will need to be managed at run time. There is work on integrating various verification tasks into self-adaptation as a way to provide assurances as an intimate part of self adaptation.

3.1 Evidence Types for Use in Assurance Case Decomposition

When considering decomposition, it is necessary to identify what kinds of evidence and sub-goals could be used in assurance cases. In this section, we summarize some of the on-going research into this, organizing it into a discussion of work that could be used for evidence from each of the MAPE-K activities.

Monitoring. Casanova [13, 14] provides some foundational work that can be used to provide evidence that there is sufficient knowledge to diagnose a problem in a system. In [13], the information theoretic concept of entropy is used to determine how much information is needed by a statistical and model driven diagnostic approach. The amount of information needed can be updated autonomously. In [14], formal criteria are given for establishing the maximum theoretical accuracy bounds for diagnostics given a set of observations, and the minimum bounds for accuracy (in the case of single fault systems). This theory can be used to provide evidence that enough monitoring is in place for specific diagnosis, even when some of the components in the system cannot be observed but might be the cause of problems.

Requirements may also be used to provide evidence of sufficient monitoring. In [1] they use contextual goal models to identify monitoring requirements. Different contexts provide different facts that can be monitored, and can lead to problems in the monitoring requirements (such as redundancy and inconsistency). SAT-solvers are used to produce equivalent and less costly monitoring requirements that can be shown to be optimal for monitoring the requirements.

The above approaches show that it is possible to provide evidence that a self-adaptive system is *monitoring enough information* about the target system. However, assurances related to the granularity, timing, and effect of the monitoring on the target system have yet to be investigated.

Analysis. The analysis activity is related to interpreting information gained from monitoring in the context of whether an adaptation should be done. Analysis can range from checking the correctness of the model to applying mathematical analysis of the model. For example, [39] uses performance and workflow models to analyse traffic coming into a distributed transactional system. These models are used at design time to determine initial resource provisioning given assumptions about the environment, and at run time to determine correct provisioning to new workflows and configurations. A similar approach is used [2] to deal with Denial of Service attacks, where performance models are used to determine whether to divert traffic to a checkpoint, which then issues a challenge to determine if the traffic originates from a bot. In this case the soundness of the mathematical models and their updating and

use at run time provide evidence that performance goals will be met. Performance models were also used in [19] to provide evidence that appropriate constraints, monitors, and mitigations are designed into a self-adaptive system to manage performance.

In general, assurances providing sound analytical models or simulations (e.g., [26]) can be used to provide evidence that analysis is sufficient to decide if some adaptation needs to be performed. One thing to note here is that parts of the analysis can be done at design time, and parts at run time. Furthermore, if the part of the analysis that is done at run time needs to be done in a timely manner, and in this case evidence needs to be given that the analysis will, for example, detect problems in enough time to do something about them.

Planning. As mentioned above, self-adaptive systems must produce adaptations that return systems from an undesired state to a normal state. Ideally, we would like to be able to provide evidence that adaptations always achieve this transition, but because of various sources of uncertainty, this is not possible. However, there has been some work that is making strides in providing some evidence. In the case where adaptations are being chosen from a set that is predefined, probabilistic model checking can be used to determine the adaptation that has the higher likelihood of success. In [43], the authors show how this approach may be used to provide evidence about the effect of adaptations on system quality objectives, and how this may be used to provide evidence that all states in some region of the state space of the system will be improved by selected adaptations. This evidence helps to show that strategies have sufficient coverage of some part of the state space. Adaptive CTL (AdaCTL) is defined in [22] to also provide some analytical evidence that there is sufficient coverage of adaptations to achieve desired goals in changing, but enumerated, environments. The summary of formal methods used in self-adaptive systems also concentrates on assurances for planning [49]. They note that many of the formal methods used for assuring planning happen during the design of the self-adaptive system, and not at run time. Filieri [24, 25], on the other hand, describe a number of strategies for using probabilistic model checking at run time for self-adaptive systems where goals are expressed as temporal logical formulas, including state elimination and algebraic approaches for making it more tractable at run time.

Execution. When an adaptation is triggered, we want to be able to assure several things. For example, we want to assure that the system correctly executes the effects and that the system and model are (eventually) consistent, we want to be able to assure that, if two adaptations can execute simultaneously, that they do not interfere with each other in unpredictable ways, and we require assurances that the execution will not in fact have a deleterious affect on the qualities that we are concerned about. Some work has been done in trying to answer these questions. In Veritas [27] some of these assurances can be evidenced by run-time testing. As the system is adapting, so too should the test cases. Veritas uses a genetic algorithm approach to evolve test cases, and utility functions to choose and prioritize test cases that need to be run to assure that the system is still executing safely and correctly.

3.2 Assurance Composition Based on the MAPE-K Loop

One of the challenges for providing assurances for self-adaptive systems is how to integrate assurances into the process of self-adaptation at run time. In the context of composition, this can be seen as developing techniques to allow evidence to be collected and collated at run time. For example, Tamura et al. [45] discuss the need for validation and verification (V&V) in self-adaptive systems, and argue that run time V&V tasks should be integrated into the activities of self-adaptation. They integrate the V&V tasks into the MAPE-K loop.

While this approach does not provide any specific techniques for providing assurances, it does define a framework for integrating and positioning self-adaptive elements that could provide some evidence for assurance cases, and could be used to structure this evidence in assurance cases. In particular, service level agreements can be thought of as high level goals in an assurance case, and so the Runtime Validator and Verifier, in checking that after execution of an adaptation the goals will be met. It is conceivable that other pieces of evidence (such as evidence that a model accurately reflects the system being managed) could be incorporated into this structure.

Another aspect of V&V discussed in [45] is viability zones, which are the set of possible systems states in which goals can be achieved that evolve with environment and context changes. This is elaborated upon in [49], which identifies adaptation zones as a way to understand the state space of self-adaptive systems, and as a framework for understanding the use of model checking in providing evidence of self-adaptive behaviour. They identify four zones: (1) Normal behaviour, which is the state where the system is running in its designed functionality; (2) Undesired behaviour, which is where a system is not meeting its goals or properties and requires adaptation; (3) Adaptation behaviour, which is when the system is adapting itself to fix the undesired behaviour, and (4) Invalid behaviour which are behaviours that the system should never exhibit (e.g., system deadlock). They then identify model checking work that assures properties in each and between these zones. This work can be used to identify the evidence types that have been used for assuring different parts of self-adaptation, and organizing such evidence around these parts that can aid in composition.

Model-checking evidence for the most part concentrates on assuring the design of the self-adaptive system. In [40], the authors outline an approach to testing implemented self-adaptive systems. They use a Failure Mode and Effects Analysis on the activities in the MAPE-K loop that allows them to categorize different possible problems that need to be assured (in this case, tested). The seven categories that they identify range from providing assurances that sensor information is correctly interpreted to assuring that adaptation effects are correctly effected in the system. All of these categories need to be assured for model based testing to be considered comprehensive. This work provides another way to organize the assurance activities that could be used to compose an assurance case for a particular goal.

4 Decomposing and Composing Assurances to Self-Adaptation

As we have argued in this chapter, we need an approach that organizes assurance techniques and the results they produce into a rational argument where justifications of the goals for the self-adaptive system can be checked and assessed throughout the system’s life cycle. In this section we discuss some of the challenges with decomposition and composition of assurances. The challenges for decomposition and composition discussed below constitute the set of requirements any approach must manage.

4.1 Decomposition of Assurances

Assurance cases decompose assurance problems by breaking high level goals into sub-goals for which it easier to provide evidence. This evidence can be combined through argumentation and judgement to provide confidence that the goals will be met. Not surprisingly, we can use assurance cases as a guide for thinking about decomposition of assurances for self-adaptive systems. In the previous section, we saw how existing work in assurances for different activities of MAPE-K can be used as evidence. In this section, we describe some of the challenges associated with decomposing the assurances.

Decomposition of Goals. A fundamental tenet of assurances cases is being able to decompose goals. In goal-oriented requirements engineering [46], goals describe the objectives that the software system should achieve. Such goals are used in the requirements engineering process for “eliciting, elaborating, structuring, specifying, analysing, negotiating, documenting, and modifying requirements” [46, p. 249]. An abstract goal (i.e., the root of a goal tree) is systematically and iteratively refined to subgoals until each subgoal (i.e., the leaves of the goal tree) can be satisfied by a set of tasks a single agent can perform. Such an agent can be a human or a software component. Approaches extending the principles of goal-oriented requirements engineering have been proposed to address self-adaptive software systems [15, 44].

For *functional* decomposition of goals, the system is decomposed into multiple components. For each component we have to establish evidence that it correctly realizes its tasks to achieve the subgoal assigned to it. For instance in Znn (c.f. Sect. 2.3), we may provide evidence for the correct behaviour and processing time of the dispatcher regardless of the size and configuration of the server pool, that is, whether the dispatching of requests works and how much time it takes. This evidence focuses on an individual component of the system and the related subgoal but contributes to the assurance of the overall response time goal of the system. Hence, functional decomposition can be exploited to decompose assurances and establish evidence for components or subsystems.

To decide whether and how to decompose *extra-functional* goals, on the other hand, is more challenging. Unlike functional goals, which can more easily be

decomposed into somewhat independent pieces, extra-functional goals are cross cutting with many interdependencies, for example resource usage, which is split over many functions. Extra-functional goals may be decomposed if they are orthogonal/independent from each other, or if the interdependencies can be managed. The latter requires knowledge about how the goals affect each other and how the goals can be balanced. Utility functions are one means to do that [17, 34]. An example from the Znn case is the response time goal that may be assured without considering the associated costs and using simulation or predictive analysis. However, this may result in over-provisioning, which is not desired. Hence, we have to consider both concerns, response time and costs, together. The question then arises whether each of them can be completely assured independently and whether the resulting assurances can be composed afterwards (cf. Sect. 4.2). The composition might require further assurances to obtain the required confidence that a composition works. If two or more goals are tightly coupled with each other, it might not be reasonable to decompose and assure each of them individually. In such cases, we may need to keep them together in the decomposition structure. In such situations assurances are not provided for leaf goals in the goal tree but rather for a subtree. The same holds for the self-management of Znn. Considering the requirements of performing an adaptation safely and within a certain time (c.f. Goal 4 in Fig. 3), we may establish evidence for both aspects individually. However, it is conceivable that executing a guaranteed safe adaptation takes more effort and time than an ad hoc adaptation. Hence, both requirements must be jointly handled when constructing and assuring them.

Decomposition Strategies Specific to Self-Adaptation. In the previous section, we discussed how we might use goal decomposition to decompose assurances, and in Sect. 3.2 we discussed how existing self-adaptive techniques might be considered as evidence in a decomposition. Another way to consider self-adaptation in the role of assurances is as a technique itself for achieving some goal in the system, and in such cases we need to provide assurances for the self-adaptation mechanism itself.

The performance goal of Znn is achieved by a controller automatically adapting the Znn architecture shown in Fig. 1 by scaling up and down the number of web servers in response to the varying load. As depicted in Fig. 3, we simplify the problem and consider only two versions of Znn, one with a smaller pool of web servers for normal load and one with a larger pool for high load. If we install a self-adaptation mechanism on top of Znn, that is, a controller that automatically reconfigures the architecture of Znn by switching between the version, we must provide assurances for the controller and the controller's interface to Znn and the environment. The addition of a self-adaptation strategy requires that we provide evidence for the strategy-specific goals (c.f. Goal 4 in Fig. 3) in the argumentation structure. This calls for a further decomposition of assurances concerning the self-adaptation mechanism. To guide this decomposition, we have identified strategies that are specific to self-adaptive software systems.

We propose that the decomposition of goals, either functional or extra-functional ones, and the identification of evidence types and techniques are guided by the reference model for self-adaptive systems depicted in Fig. 4. It provides an architectural perspective on self-adaptive systems and is helpful to identify architectural concerns for self-adaptation that require assurances and that should be included in the argumentation structure.

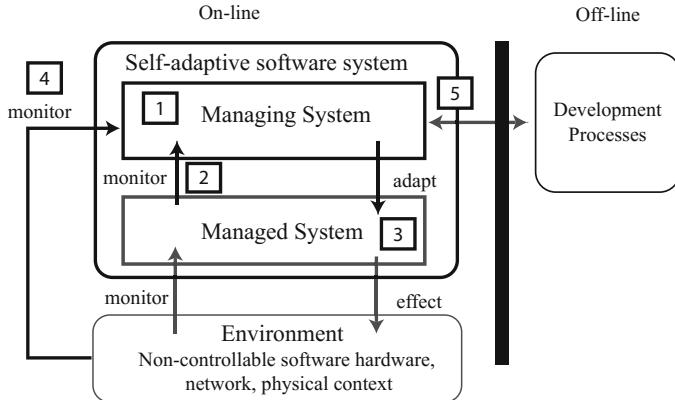


Fig. 4. Architectural reference model for self-adaptive software systems.

A managing system monitors the managed system and the environment to make a decision about adapting the managed system if the goals are not satisfied or if their satisfaction steadily decreases. For instance, in the Znn example, a controller monitors Znn to observe the current architecture and the response time, and it monitors the network to observe the number of connected clients as an indicator for the current load imposed on Znn. If the monitored response time violates the performance goal, the controller decides about scaling up the web servers and the number of web servers to be added, which is eventually translated to Znn by executing this adaptation. Based on Fig. 3, the controller would switch to the Znn version designed for the high load. Finally, this architectural reconfiguration of Znn that switches to the high-load version should bring back the system into a state that fulfils the performance goal.

Consequently, besides providing assurances for the managed system such as Znn, we have to establish assurances for the managing system. We exemplify this with its functional goals, where we must provide convincing evidence that the managing system

- makes a correct decision of when and how to adapt the managed system (cf. **1** in Fig. 4),
- correctly monitors the managed system **2** and the environment **4** and that the assurances and assumptions provided for managed system and environment are correct such that the managing system can rely on them,

- correctly adapts the managed system **[3]** that in turn must change according to this adaptation,
- correctly interacts with the development process **[5]**, for example, an administrator directly adapts the running Znn instance in a situation that the managed system cannot handle, or an engineer tunes the adaptation strategies of the managing system to improve the performance of the self-adaptation.

A similar exercise for the managing system's extra-functional goals can be guided by the same reference architecture. Using this approach, the managing and managed systems do not have to be considered as black boxes and their decompositions can be taken into account. We may repeat the decomposition and refine the managing system to monitor, analyse, plan, execute, and knowledge components as proposed by MAPE-K [38] and consequently, assurances can be provided for the individual components rather than for whole controller.

These aspects (**[1]** to **[5]**) must be covered by assurance cases. For the performance goal of Znn, evidence is required that Znn properly implements the monitor **[2]** and adapt **[3]** interface such that the controller can rely on certain timeliness and accuracy of monitored data and on the eventual execution of a reconfiguration. Moreover, evidence is required that the assumption concerning the environment **[4]** hold, for instance, that the controller can reliably derive the user load on Znn from the network. Finally, evidence is needed that the controller itself works properly **[1]**. For instance, a controller typically should fulfil the properties of stability, accuracy, settling time, and overshoot [32] in addition to maintaining the managed system in a state that fulfils the goals such as the performance goal in the case of Znn.

Challenges. So far, we have outlined how we might use assurance cases and the MAPE-K pattern as a framework for decomposing assurances. We now summarize the challenges.

Time- and Lifecycle-related decomposition: The connection between on-line and off-line assurances (cf. Fig. 4) raises a number of challenges. On-line techniques are embedded in the self-adaptive system while off-line techniques work in the development or maintenance environment of the system. Though both kinds of techniques are used while the system is running, the distinction becomes relevant if costly assurance techniques such as model checking cannot be used on-line and thus have to be performed off-line. One challenge here is to understand which evidence and goals are more suitable for run time collection and verification, and which are more suitable earlier. For example, providing evidence through model checking tools is difficult to do at run time because of state explosion and computational complexity, but doing this earlier may not account for uncertainty. In this case, we need to explore ways to parameterise the model checking so that parts of it can be done at run time and parts at design time. But a general challenge is how to use evidence collected before deployment to inform and make efficient any run-time analysis. We also need to develop guidelines of how to decompose the evidence along the time dimension.

Matching evidence with goals: Decisions have to be made about which evidence types and techniques should be used for the assurance of which goals, and how we know that we have enough evidence to assure a goal. This requires knowledge about the different evidence types, such as the level of confidence that they provide (e.g., simulation results refer to individual traces of the system while model checking results refer to the whole state space of the system) or the costs of using them (e.g., model checking can be infeasible due to the problem of state-space explosion).

Assurance additivity and independence: During decomposition of assurances, assurance cases are mostly assumed to be global to the system, and so at the top level are assumed to be independent and complete — conflicts are resolved with the goal tree and rationale. For self-adaptive systems, this global assumption will not hold if we are composing goals and systems at run time. In this case, we need to be able to reason about assurance additivity, independence, and conflict resolution. The identification of assurances that remain independent and can be added together is important for composition — in this case, composition is relatively straightforward. An important aspect of this challenge is to develop a set of sufficient criteria for assurance independence. Part of this challenge may be alleviated with strong provenance and annotations of global assumptions made during decomposition. At the systems of systems level, the subsumption of goals and conflicts in goals must also be identified. For example, if we have an assurance case in Znn for a goal of the response time being less than 2 s, and we are composing with an assurance for a goal of less than 5 s, is the former assurance case sufficient?

4.2 Composition of Assurances

An orthogonal approach to decomposition is *composition*. While the aim of decomposition is to make the task of gathering individual assurances simpler via modularization, composing assurances aims to construct an argument by assembling arguments together. To achieve composition, global system information is required. Decomposition may remove the larger context of the system, however it is necessary for each assurance to maintain its provenance and running context, as well as have a clear position within the argumentation structure. As mentioned in Sect. 4.1, individual decomposed assurance cases have a close analogy with unit tests, while composing assurances is more closely aligned with integration and system testing. Unit tests are run without the global view of the system, and may either over- or under-approximate system behaviour, while integration and system tests consider the environment under which they are run. Bate and Kelly [3] argue that to compose assurances for modular systems the modules should align with the hardware and software components. They also point out the need to consider trade-offs in goals which we discuss below. However, if decomposition should be performed via goals and evidence, then we argue that composition should follow these dimensions as well. Our discussion below assumes that we take this view of composition.

Types of Composition. In composition of assurances (either goal-based or evidence-based), individual facts are aggregated to confirm that a goal holds. Composition can be of three types (1) composing assurances from a single system with a single set of goals, (2) composing two or more individual systems each with their own goals (and assurance cases), or (3) composing systems of systems, with multiple systems, multiple goals and multiple assurance units for each. In the first type, if our argumentation structure from the original decomposition exists, then composition can simply gather the individual assurances using the provided argumentation structure and compose these with confidence. We focus instead on challenges that arise due to the second and third type of composition.

When composing assurances from two different systems that work together (type 2), each may have its own unique goals. Consider the case where we introduce a second system, Zbay, to work in concert with Znn that requires a higher security profile than the original to permit financial transactions. Zbay runs over an https connection and has a less stringent QoS goal from the original insecure Znn — the response time must be less than five seconds. It also has additional goals not found in Znn, related to its security requirements. If we want to assure that these two systems can work in coordination, we will need to compose their assurance cases. The assurance cases for G2 and G3 from Znn and similar ones for Zbay now have different sub-goals, and their composition depends on which system they are assured under. If, for instance, we assure G2 under Znn and G3 under Zbay, there is no guarantee that these will still satisfy G1 for Znn when combined. However, this composition should suffice for Zbay and in fact, the composition of G1 and G2 from Znn could be argued to be sufficient for the whole system composition, if the argumentation structure can show that the http connection is always at least as slow or slower than https. If we now consider the context further, the dispatcher, which was previously assumed to be independent under the Znn argumentation structure, may no longer be independent in this larger system. It is possible that dispatching across http protocols changes some global assumptions and a new argumentation structure may be needed. Additionally, for the security goals, they are found only in the Zbay system, but since Zbay now can send information through Znn, the Znn goals may need to be revised to assure that sensitive information cannot flow from Znn to Zbay. We may also find dependencies between systems and goals that must be added to the argumentation structure. If, for instance, we are under https, it is possible that another aspect of the system is disallowed (such as ftp). Arguments from Znn that include this protocol must now be revisited in this larger context.

Composing two individual systems has its challenges, but as we allow for an arbitrarily large number of systems, the challenges increase. In general we see this as a systems of systems view of composition (type 3). Under this scenario, we may have multiple variants of Znn and Zbay, such as Zamazon, Zxpedia, etc. Each of these systems has a set of common goals, and may even share components such as the dispatcher. Yet they each also have their own unique requirements, working environments and constraints. To ensure that these systems can work together, we must combine assurances across the entire system. This leads to a

potential combinatorial explosion in the number of compositions that can occur between systems. Not only do we now have to face the problem of combining two assurances, we may find complex interactions between three or more assurances, and it will become infeasible to validate all compositions. There may be dependencies as well, either within the systems themselves, or ones that are global. Unlike the type 2 constraints these may now span the entire system. To assure systems of this type, we may need to resort to a sampling scheme (such as that used in combinatorial testing [20]), and accept that our argumentation only provides a certain level of assurance across the system, rather than a comprehensive one. For instance, we may argue that we know all combinations of pairs of assurances can be composed, but we may not be able to guarantee that combinations of a higher arity of assurance is still valid. Another issue that arises in systems of systems is that of competing assurances. For instance, in Zbay and Zamazon the need for security may be more important than the goal of a low response time, however in Zxpedia and Znn, response time may be paramount. Some sort of weighted utility is possible, or we may to allow for multiple solutions for a goal and view this as a Pareto front to understand the trade-offs in time and security goals.

One possible way to model and simplify the view of a self-adaptive software system is as a set of *features* that are added and removed as the system adapts [23, 29]. Elkhodary et al. first presented the notion of using features for directing adaptation for QoS aspects of a system [23]. Garvin et al. also suggested using this view of adaptation, but from a more traditional functional view of features [29]. Software product line engineering [23] provides many tools that may help our reasoning and analysis, both from a goal based and from an evidence-based view. We can then use these models to describe composition and sampling and to guide our argumentation structure.

Challenges. In the above, we outlined some challenges particular to different types of composition. We now summarize the challenges for composing assurances in general.

Time-based Composition: The time (or the state at which an assurance is obtained) can change the outcome of the evidence, or may change the type of evidence to be gathered. If Znn and Zbay are implemented as services, then the types of evidence available for composition can vary at run time, depending on which services are currently active. For instance, the dispatch service may have different variants and within those variants use different mechanisms. If we assure the system under one variant of the dispatcher but later compose our system using a different variant of the dispatcher, the original assurances may not hold. This problem can occur in all three types of composition (1–3). This dynamic view of composition leads to a new level of complexity and may require a new argumentation structure; one that was not considered during decomposition.

Assurance dependencies: In related work on software testing for component-based or configurable systems, determining which features are dependent on

others has proven to be challenging [21]. Documentation is often lacking and therefore this must be performed by domain experts, and/or via program analysis. For assurance case composition this is even more challenging. Which strategies depend on particular evidence types and how does that relate to the overall assurance case? If a composition results in reusing evidence in multiple assurances cases, how do we keep track when those goals change? How do we find the dependencies and goals that need to be added to argumentation structures?

Evidence reuse: In non-adaptive systems, evidence may be reused to support multiple assurance cases. The same kind of reuse is less obvious when self-adaptation is involved. For example, if evidence for load balancing under a certain set of activated self-adaptations is collected, it may apply under a different set of adaptations, or it may not. The larger space of potential system states makes reusing assurances more challenging.

5 Applying Assurance Cases to Self-Adaptation

In the previous sections, we outlined how we might structure decomposition and composition of assurances for self-adaptive systems, and highlighted some of the challenges with each of these. In this section we describe some emerging work in assurance cases composition and decomposition that could be applied to help make assurance cases more useful for self-adaptive systems.

5.1 Assurance Case Decomposition and Composition Research

Safety Case Patterns. While reasoning about satisfaction of individual goals using assurance cases is useful, a means of reusing and combining assurance cases is required. Some studies in developing goal-based approaches aim to support the reuse and modularization of safety cases so that safety arguments for sub-systems/components can be re-used in other contexts. The notion of safety case patterns [33] can be applied in order to explicitly model common elements found between various safety cases created for particular applications. Patterns in arguments can emerge, for example, typical combinations of arguments and accepted interpretations of specific types of evidence. These can be documented by means of a safety pattern language. This solution promotes a structured reuse of the safety case rationale instead of its informal material reuse.

More recently, the work by Hawkins et al. [31] has defined a safety argument pattern catalogue in order to guide developers in structuring *maintainable* safety arguments. The idea is to provide evidence for low-level claims, considering different levels of abstraction suitable for different stakeholders of the system. The assumption is that as the software system moves through the development life-cycle, there are numerous assurance considerations against which evidence must be provided.

Such patterns could help guide the kind of evidence that needs to be collected, or how to structure assurance arguments for particular goals of the system.

Modularization and Contracts. The work by Ye and Kelly [51] proposes the use of contracts to modularize safety cases in order to capture application-specific safety requirements, and corresponding assurance requirements derived for a potential COTS (common-off-the-shelf) component. This contract can be used to form the basis of a safety case module for the component. The notion of compositional safety case construction proposed by Kelly [36] is used for modelling the safety case of the application separated from the assurance requirements of the component. More generally, system safety cases are often decomposed into sub-system safety cases to cope with their complexity. As a consequence, GSN was extended with the notion of UML packages and “Away Goals” in order to support the notion of modular safety case construction. Moreover, as pointed out in [36], the need for a modular safety approach is becoming more apparent when considering new types of modern systems that are emerging, such as systems of systems [3]. For self-adaptive systems, the notion of contracts could be useful in reasoning about the composition of assurance cases.

Decomposition and Composition. The support for claim decomposition and structuring is very informal and argumentation is seldom explicit [4]. In practice, the emphasis is on communication and knowledge management of the safety cases, with little guidance on what claim or claim decomposition should be performed. Some studies are developing more rigorous approaches to claim decomposition in order to demonstrate that the decomposition is complete, that is, that the sub-claims demonstrate the higher claim [4]. Furthermore, the authors highlight the importance of (i) more efficient means for modelling safety cases since they are costly to develop, and (ii) improving safety case structuring to provide safety case modularization, to use diverse arguments and evidence, and to exploit the relationship between the argument structure and the architecture of a system.

The work by Voss et al. [47] also explores the idea of modular certification when reusing components from one system to the next, that is, when reusing a system element, engineers can (in)formally reuse the associated safety arguments of the element. This solution supports a component-based development process and a model-based tool to specify the system’s architecture at different layers of abstraction and it integrates the construction of the system and the argumentation about its functional safety. Of course, increased rigour in claim decomposition could be exploited for assuring self-adaptive systems.

5.2 Challenges Applying Assurance Cases to Self-Adaptation

This chapter has taken the position that work in assurance cases can be used to guide the decomposition and composition of assurances for self-adaptive systems. While we believe that this is a good approach, there are distinct challenges with applying assurance cases to self-adaptive systems.

Uncertainty: Self-adaptive systems are often self-adaptive because they are deployed in environments with uncertainty. This uncertainty affects the types

of evidence that can be collected to support assurances, the ways in which the evidence can be collected, and even the specification of the assurance case itself. For example, goals in assurances cases need to specify the environmental assumptions under which they are valid but for self-adaptive systems we need some way to make uncertainty about these assumptions first-class.

Adaptation assurances: When conditions change and the system adapts, an assurance may describe how quickly or how well it adapts. For example, with Znn, an increased demand may trigger the addition of a web server. An assurance may state that when the per-server load exceeds a threshold, the system adapts within two minutes by adding web servers and the per-server load falls below the threshold within five minutes. This assurance may hold at all times, or may be expected to hold only when the demand increases but then remains constant.

Automatable assurance cases: As mentioned in Sect. 2.2, assurance cases rely on human judgement to discern whether the argument and rationale actually makes the case given the evidence. One of the aims of self-adaptation is to eliminate or at least reduce the involvement of humans in the management of a software system. To accomplish this, self-adaptation requires ways to computationally reason about assurance cases, and a logic to judge whether an assurance case is still valid, what changes must be made to it in terms of additional evidence, etc.

Adaptive assurances: As just alluded to, self-adaptation may require the assurance cases themselves to adapt. For example, replacing a new component into the system may require replacing evidence associated with that component in the assurance case. Changing goals of the system based on evolving business contexts will likely involve changes to the assurance cases for those goals. Automatable assurance cases are an initial step to addressing this challenge, but approaches, rules, and techniques for adapting the assurance cases themselves are also needed.

Assurance processes for self-adaptive software systems: One overarching challenge is the design of adequate assurances processes for self-adaptive systems. Such a process connects the system's goals, the architecture, and implementation realizing the goals to the assurance cases' argumentation structures, its strategies, evidence types, and assurance techniques. This challenge requires that parts of the design and assurance process that was previously performed off-line during development time must move to run time and carried out on-line in the system itself. The assurance goals of a system are dependent on a correct, efficient and robust assurance process, which employs on-line and off-line activities to maintain continuous assurance support throughout the system life cycle. Currently, such processes are not sufficiently investigated and understood.

Reassurance: If we are able to move the evaluation of assurance cases to run time, then challenge arises in how to reassure the system when things change. Reassurance may need to happen when environment states, or the state of the system itself, change. Which part of the assurances case needs to be re-evaluated? For composition, where the composition itself is dynamic, we need ways to identify

the smallest set of claims (goals) that have to be reassured when two systems are composed? Which evidence needs to be re-established, and which can be reused?

6 Conclusions

We have considered the challenges associated with decomposing and composing assurances for self-adaptive systems. While there is a large body of work in software assurance that is beginning to address this for general software systems, self-adaptive systems raise further inherent challenges.

We have discussed assurance cases as an approach to reasoning about composing and decomposing assurances for self-adaptive systems. Assurance cases provide a discipline for decomposing assurances in a principled way. Furthermore, there is some work related to assurance cases also addresses assurance composition, meaning that assurance cases may also be suitable for reasoning about composition of assurances for self-adaptive systems, and also for composing assurances in self-adaptive systems of systems. We believe that applying assurance case approaches to the problem of assuring self-adaptive systems shows great promise.

At the same time, there are many aspects of self-adaptive systems that present challenges to assurance case research. We provided some of these challenges in Sect. 5.2. Most of these challenges arise from the need for self-adaptive systems to respond to changes in the environment or the requirements of the system, and so we need ways to assess elements of assurance cases automatically, and evolve them, at run time.

References

1. Ali, R., Griggio, A., Franzén, A., Dalpiaz, F., Giorgini, P.: Optimizing monitoring requirements in self-adaptive systems. In: Bider, I., Halpin, T., Krogstie, J., Nurcan, S., Proper, E., Schmidt, R., Soffer, P., Wrycza, S. (eds.) BPMDS/EMM-SAD -2012. LNBP, vol. 113, pp. 362–377. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31072-0_25
2. Barna, C., Shtern, M., Smit, M., Tzerpos, V., Litoiu, M.: Mitigating dos attacks using performance model-driven adaptive algorithms. ACM Trans. Auton. Adapt. Syst. **9**(1), 3:1–3:26 (2014)
3. Bate, I., Kelly, T.: Architectural considerations in the certification of modular systems. In: Anderson, S., Felici, M., Bologna, S. (eds.) SAFECOMP 2002. LNCS, vol. 2434, pp. 321–333. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45732-1_31
4. Bloomfield, R., Bishop, P.: Safety and assurance cases: past, present and possible future - an Adelard perspective. In: Dale, C., Anderson, T. (eds.) Making Systems Safer, pp. 51–67. Springer, London (2010). https://doi.org/10.1007/978-1-84996-086-1_4
5. Bloomfield, R., Peter, B., Jones, C., Froome, P.: ASCAD – Adelard Safety Case Development Manual. Adelard, 3 Coborn Road, London E3 2DA, UK (1998)
6. Brun, Y., Bang, J.Y., Edwards, G., Medvidovic, N.: Self-adapting reliability in distributed software systems. IEEE Trans. Softw. Eng. (TSE) (2015) (in press)

7. Brun, Y., Edwards, G., Bang, J.Y., Medvidovic, N.: Smart redundancy for distributed computation. In: Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS), Minneapolis, MN, USA, pp. 665–676, June 2011, <https://doi.org/10.1109/ICDCS.2011.25>
8. Brun, Y., Medvidovic, N.: Fault and adversary tolerance as an emergent property of distributed systems' software architectures. In: Proceedings of the 2nd International Workshop on Engineering Fault Tolerant Systems (EFTS), Dubrovnik, Croatia, pp. 38–43, September 2007, <https://doi.org/10.1145/1316550.1316557>
9. Brun, Y., Medvidovic, N.: An architectural style for solving computationally intensive problems on large networks. In: Proceedings of Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Minneapolis, MN, USA, May 2007, <https://doi.org/10.1109/SEAMS.2007.4>
10. Brun, Y., Medvidovic, N.: Keeping data private while computing in the cloud. In: Proceedings of the 5th International Conference on Cloud Computing (CLOUD), Honolulu, HI, USA, pp. 285–294, June 2012, <https://doi.org/10.1109/CLOUD.2012.126>
11. Brun, Y., Medvidovic, N.: Entrusting private computation and data to untrusted networks. *IEEE Trans. Dependable Secure Comput. (TDSC)*, **10**(4), 225–238 (2013), <https://doi.org/10.1109/TDSC.2013.13>
12. Brun, Y., Reishus, D.: Path finding in the tile assembly model. *Theoret. Comput. Sci.* **410**(15), 1461–1472 (2009), <https://doi.org/10.1016/j.tcs.2008.12.008>
13. Casanova, P., Garlan, D., Schmerl, B., Abreu, R.: Diagnosing architectural runtime failures. In: Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 20–21 May 2013 (2013)
14. Casanova, P., Garlan, D., Schmerl, B., Abreu, R.: Diagnosing unobserved components in self-adaptive systems. In: 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Hyderabad, India, 2–3 June 2014 (2014)
15. Cheng, B.H.C., Sawyer, P., Bencomo, N., Whittle, J.: A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 468–483. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04425-0_36
16. Cheng, B.H.C., et al.: Software engineering for self-adaptive systems: a research roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_1
17. Cheng, S.-W., Garlan, D., Schmerl, B.: Architecture-based self-adaptation in the presence of multiple objectives. In: Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Shanghai, China, 21–22 May 2006 (2006)
18. Cheng, S.-W., Garlan, D., Schmerl, B.: Evaluating the effectiveness of the rainbow self-adaptive system. In: Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2009), Vancouver, BC, Canada, May 2009
19. Cheng, S.-W., Garlan, D., Schmerl, B., Sousa, J.A.P., Spitznagel, B., Steenkiste, P.: Using architectural style as a basis for self-repair. In: Bosch, J., Gentleman, M., Hofmeister, C., Kuusela, J. (eds.) Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture, 25–31 August 2002, pp. 45–59. Kluwer Academic Publishers (2002)
20. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: an approach to testing based on combinatorial design. *IEEE Trans. Software Eng.* **23**(7), 437–444 (1997)

21. Cohen, M.B., Dwyer, M.B., Shi, J.: Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach. *IEEE Trans. Software Eng.* **34**(5), 633–650 (2008)
22. Cordy, M., Classen, A., Heymans, P., Legay, A., Schobbens, P.-Y.: Model checking adaptive software with featured transition systems. In: Câmara, J., de Lemos, R., Ghezzi, C., Lopes, A. (eds.) *Assurances for Self-Adaptive Systems*. LNCS, vol. 7740, pp. 1–29. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36249-1_1
23. Elkhdary, A., Esfahani, N., Malek, S.: FUSION: a framework for engineering self-tuning self-adaptive software systems. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2010, pp. 7–16 (2010)
24. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: 33rd International Conference on Software Engineering (ICSE), pp. 341–350, May 2011
25. Filieri, A., Tamburrelli, G.: Probabilistic verification at runtime for self-adaptive systems. In: Câmara, J., de Lemos, R., Ghezzi, C., Lopes, A. (eds.) *Assurances for Self-Adaptive Systems*. LNCS, vol. 7740, pp. 30–59. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36249-1_2
26. Franco, J., Correia, F., Barbosa, R., Zenha-Rela, M., Schmerl, B., Garlan, D.: Improving self-adaptation through software architecture-based stochastic modeling. *J. Syst. Softw.* **42**(1), 75–99 (2016)
27. Fredericks, E.M., DeVries, B., Cheng, B.H.C.: Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty. In: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, pp. 17–26. ACM, New York (2014)
28. Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., Steenkiste, P.: Rainbow: architecture-based self adaptation with reusable infrastructure. *IEEE Comput.* **37**(10), October 2004
29. Garvin, B.J., Cohen, M.B., Dwyer, M.B.: Failure avoidance in configurable systems through feature locality. In: Câmara, J., de Lemos, R., Ghezzi, C., Lopes, A. (eds.) *Assurances for Self-Adaptive Systems*. LNCS, vol. 7740, pp. 266–296. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36249-1_10
30. Goal Structuring Notation (GSN) community standard version 1, November 2011, <http://goalstructuringnotation.info>
31. Hawkins, R., Clegg, K., Alexander, R., Kelly, T.: Using a software safety argument pattern catalogue: two case studies. In: Flammini, F., Bologna, S., Vittorini, V. (eds.) *SAFECOMP 2011*. LNCS, vol. 6894, pp. 185–198. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24270-0_14
32. Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D.M.: *Feedback Control of Computing Systems*. Wiley, Chichester (2004)
33. High, K.M., Kelly, T.P., Mcdermid, J.A.: Safety case construction and reuse using patterns. In: 16th International Conference on Computer Safety and Reliability, SAFECOMP 1997, pp. 55–69. Springer, London (1997). https://doi.org/10.1007/978-1-4471-0997-6_5
34. Huber, N., Hoorn, A., Koziolek, A., Brosig, F., Kounev, S.: Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments. *SOCA* **8**(1), 73–89 (2014)
35. Jennings, N.R.: An agent-based approach for building complex software systems. *Commun. ACM* **44**(4), 35–41 (2001)

36. Kelly, P.: Managing complex safety cases. In: Redmill, F., Anderson, T. (eds.) *Current Issues in Safety-Critical Systems*, pp. 99–115. Springer, London (2003), https://doi.org/10.1007/978-1-4471-0653-1_6
37. Kelly, T., Weaver, R.: The goal structuring notation - a safety argument notation. In: *Proceedings of Dependable Systems and Networks 2004 Workshop on Assurance Cases* (2004)
38. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. *Computer* **36**(1), 41–50 (2003)
39. Litoiu, M.: A performance analysis method for autonomic computing systems. *ACM Trans. Auton. Adapt. Syst.* **2**(1), March 2007
40. Püschel, G., Götz, S., Wilke, C., Abmann, U.: Towards systematic model-based testing of self-adaptive software. In: *ADAPTIVE 2013, The Fifth International Conference on Adaptive and Self-Adaptive Systems and Applications*, pp. 65–70 (2013)
41. Puviani, M., Cabri, G., Zambonelli, F.: A taxonomy of architectural patterns for self-adaptive systems. In: *International C* Conference on Computer Science and Software Engineering, C3S2E13*, Porto, Portugal, pp. 77–85, July 2013
42. Salehie, M., Tahvildari, L.: Self-adaptive software: landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.* **4**(2) (2009)
43. Schmerl, B., Cámarra, J., Gennari, J., Garlan, D., Casanova, P., Moreno, G.A., Glazier, T.J., Barnes, J.M.: Architecture-based self-protection: composing and reasoning about denial-of-service mitigations. In: *HotSoS 2014: 2014 Symposium and Bootcamp on the Science of Security*, Raleigh, NC, USA, 8–9 April 2014 (2014)
44. Silva Souza, V.E., Lapouchian, A., Robinson, W.N., Mylopoulos, J.: Awareness requirements for adaptive systems. In: *Proceeding of the 6th International Symposium on Software Engineering for Adaptive and Self-managing Systems (SEAMS 2011)*, pp. 60–69. ACM, New York (2011)
45. Tamura, G., et al.: Towards practical runtime verification and validation of self-adaptive software systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) *Software Engineering for Self-Adaptive Systems II. LNCS*, vol. 7475, pp. 108–132. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_5
46. Van Lamsweerde, A.: Goal-oriented requirements engineering: a guided tour. In: *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering, RE 2001*, pp. 249–262. IEEE Computer Society, Washington, DC (2001)
47. Voss, S., Schätz, B., Khalil, M., Carlan, C.: Towards modular certification using integrated model-based safety cases. In: *Proceedings of VeriSure: Verification and Assurance* (2013)
48. Watanabe, T., Yonezawa, A.: Reflection in an object-oriented concurrent language. In: *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 306–315 (1988)
49. Weyns, D., Iftikhar, M.U., de la Iglesia, D.G., Ahmad, T.: A survey of formal methods in self-adaptive systems. In: *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering, C3S2E 2012*, pp. 67–79. ACM, New York (2012)

50. Weyns, D., et al.: On patterns for decentralized control in self-adaptive systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 76–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_4
51. Ye, F., Kelly, T.: Contract-based justification for cots component within safety critical applications. In: Cant, T. (ed.) Ninth Australian Workshop on Safety-Related Programmable Systems (SCS 2004). CRPIT, vol. 47, pp. 13–22. ACS, Brisbane (2004)

What Can Control Theory Teach Us About Assurances in Self-Adaptive Software Systems?

Marin Litoiu¹, Mary Shaw², Gabriel Tamura³, Norha M. Villegas^{3(✉)}, Hausi A. Müller⁴, Holger Giese⁵, Romain Rouvoy^{6,7}, and Eric Rutten⁷

¹ York University, Toronto, Canada
mlitoiu@yorku.ca

² Carnegie Mellon University, Pittsburgh, USA
mary.shaw@cs.cmu.edu

³ Universidad Icesi, Cali, Colombia
{gtamura,nvillega}@icesi.edu.co

⁴ University of Victoria, Victoria, Canada
hausi@cs.uvic.ca

⁵ Hasso Plattner Institute for Software Systems Engineering, Potsdam, Germany
holger.giese@hpi.uni-potsdam.de

⁶ University of Lille 1, Villeneuve-d'Ascq, France
romain.rouvoy@univ-lille1.fr

⁷ Inria, Saclay, France
eric.rutten@inria.fr

Abstract. Self-adaptive software (SAS) systems monitor their own behavior and autonomously make dynamic adjustments to maintain desired properties in response to changes in the systems' operational contexts. Control theory provides verifiable feedback models to realize this kind of autonomous control for a broad class of systems for which precise quantitative or logical discrete models can be defined. Recent MAPE-K models, along with variants such as the hierarchical ACRA, address a broader range of tasks. However, they do not provide the inherent assurance mechanisms that control theory does, as they do not explicitly identify and establish the properties that reliable controllers should have. These properties, in general, result not from the abstract models, but from the specifics of control strategies, which are precisely what these models fail to analyze. We show that, even for systems too complex for direct application of classical control theory, the abstractions of control theory provide design guidance that identifies important control characteristics and raises critical design issues about the details of the strategy that determine the controllability of the resulting systems. This in turn enables careful reasoning about whether the control characteristics are in fact achieved. In this chapter we examine the control theory approach, explain several control strategies illustrated with examples from both domains, classical control theory and SAS, and show how the issues addressed by these strategies can and should be seriously considered for the *assurance* of self-adaptive software systems. From this examination we distill challenges for developing principles that may serve as the basis of a control theory for the assurance of self-adaptive software systems.

1 Introduction

This chapter explores the contributions that classical feedback loops, defined by control theory, can make to self-adaptive software (SAS) systems, particularly to their assurances. To do this, we focus on the abstract character of classical feedback loops—how they are formulated, the assurances they provide, and the analysis required to obtain those assurances. Feedback loops, in one form or another, have been adopted as cornerstones of software-intensive self-adaptive systems, as documented in [33] and later on in [12,38]. Building on this, we study the relationships between feedback loops and the types of assurance they can help provide to SAS systems. We focus particularly on the conceptual rather than implementation level of the feedback model. That is, we concentrate on the relationships among desired properties that can be ensured by control theory applied in feedback loops (*e.g.*, stability, accuracy, settling time, efficient resource use), the ways computational processes can be adjusted, the choice of the control strategy, and the quality attributes (*e.g.*, responsiveness, latency, reliability) of the resulting system.

More concretely, this exploration includes, on the one hand, how feedback loops contribute to providing assurances about the behavior of the controlled system and, on the other hand, how assurances improve the realization of feedback loops in SAS. To set the stage for the discussion of concrete challenges identified in this exploration, we first review the major concepts of traditional control theory and engineering, then study the parallels between control theory [5,43] and the more recent research on feedback control of software systems (*i.e.*, MAPE-K loops) [29,33] in the realm of SAS. We establish a common vocabulary of key concepts, such as the control objective or setpoint, the disturbances that may affect the system, control responsiveness, and the control actions used by the controller to adapt the system. By discussing several types of control strategies, backed up by concrete examples, we illustrate how the control theory approach can direct attention to decisions that are often neglected in the engineering of SAS [12]. From the analysis of these concrete examples, we posit key challenges for assurance research related to the application of feedback loops in self-adaptive software.

The contribution of this chapter is twofold. First, it provides a comprehensive, and easy to understand, explanation of the application of control theory concepts to the engineering of SAS systems. This is valuable for both control engineering researchers interested in applying control engineering in new domains, and SAS researchers and engineers interested in taking advantage of control theory concepts and techniques for the engineering of SAS systems. Second, the chapter discusses assurance challenges faced by SAS designers and extrapolates control theory questions, concepts, and desirable properties into the SAS systems realm.

The remaining sections of this chapter are organized as follows. Section 2 revisits SAS foundational concepts, particularly the MAPE-K loop. Section 3 explains control theory concepts, including their analogy with the corresponding elements in SAS systems. Section 4 extends these models to adaptive and hierarchical control. Section 5 presents an overview of the application of

control theory to the self-adaptation of software systems, focusing on the model and controller definitions. Section 6 discusses classic control strategies and the assurances they provide about the quality of the control, based on control theory properties; it also analyzes the design of controllers with desirable control theory properties. Section 7 discusses relevant assurance challenges in the design of SAS systems, extrapolated from those of control theory. Finally, Sect. 8 summarizes and concludes the chapter.

2 Self-Adaptive Software (SAS) Systems

The necessity of a change of perspective in software engineering has been discussed during the last decade by several researchers and practitioners in different software application domains [16, 18, 44]. In particular, Truex *et al.* posited that software engineering has been based on the incorrect assumption that software systems should support rigid and immutable business structures and requirements, have low maintenance, and fully satisfy their requirements from the initial system delivery [42, 57]. In contrast to this “stable” vision, the engineering of self-adaptive software (SAS) originates from a different paradigm based on continuous analysis, dynamic requirements negotiation and incomplete requirements specification.

SAS systems adjust their own structure or behavior at runtime to regulate the satisfaction of functional, behavioral and non-functional requirements that vary over time, for instance when affected by changes in business goals or the system’s execution environment [16, 36, 37, 49, 55]. When system requirements evolve in this way, the system must support both short-term adaptation mechanisms to ensure satisfaction of current requirements and also long-term evolution of the requirements [42, 45]. The former requires continuous adjustments of system parameters to satisfy current requirements, and the latter requires runtime system analysis to direct long-term evolution [12, 16]. Furthermore, assurance mechanisms, both at design time and runtime, must be implemented to guarantee required properties [56].

Inspired by the human autonomous nervous system, IBM researchers proposed the *autonomic element* as a building block for developing self-managing and autonomic software systems. They synthesized this adaptation in the form of the so-called Monitoring-Analysing-Planning-Execution and shared Knowledge (MAPE-K) loop model, as depicted in Fig. 1. The purpose of this model is to develop autonomous control mechanisms to regulate the satisfaction of dynamic requirements, specifically in software systems [29, 30, 33]. An autonomic manager not only implements the phases of the adaptation process, but also provides the interfaces (*i.e.*, manageability endpoints) required to sense the environment, and to effect changes on the target system (*i.e.*, the system to be adapted, also referred as the managed application) or to manage other autonomic managers.

The responsibilities of each phase of the MAPE-K loop depicted in Fig. 1 are defined as follows:

- *Monitoring.* In this phase sensors keep track of changes in the managed application’s internal variables corresponding to desired properties (*e.g.*, measured

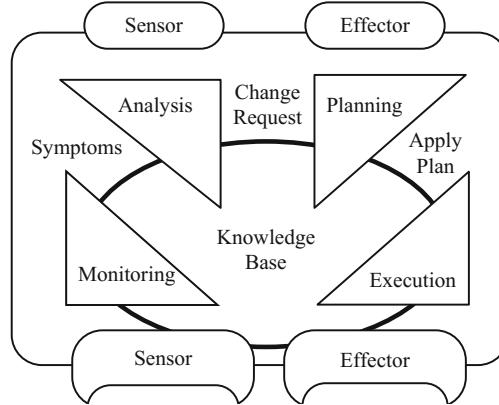


Fig. 1. The MAPE-K loop [33]

QoS data), the properties to be assured on the adaptation mechanism (*e.g.*, stability), and the external context (*i.e.*, measured from outside the managed application). Based on these changes, monitors must notify relevant context events to the analyzer. Relevant context events are those derivable from system requirements (*e.g.*, from QoS contracts). At this phase, the *knowledge base* can provide information, such as the values desired for the properties that are being controlled, and historical information that is useful for inferring context events.

- *Analysis.* Based on the high-level requirements to fulfill, the context events notified by monitors from the monitoring phase, and using the *knowledge base*, in this phase the analyzer determines whether a system adaptation must be triggered and notifies the planner accordingly. This would occur, for instance, when the notified events signal changes that (may) violate the reference control inputs.
- *Planning.* In this phase, once notified with a reconfiguration event from the context analyzer, the planner generates a strategy to fulfill the new requirements, using the accumulated knowledge in the *knowledge base*. By applying the generated strategy, the planner computes the necessary control actions to be instrumented in the managed software system as discrete operations that are then interpreted by the executor in the next phase.
- *Execution.* In this phase, after receiving a reconfiguration plan, the executor interprets each of the discrete operations specified in the plan and effects them on the managed software system. This implies translating or tailoring the reconfiguration actions to the ones supported by the platform that executes the managed software system. The *knowledge base* may provide information useful for the adaptation process, such as reconfiguration rules or runtime reference models.

3 Feedback Control

Control theory depends on reference control points of system behavior and corresponding explicit mathematical model specifications. Some researchers advocate the idea that control theory is applicable as is to SAS systems and propose a process for realizing this idea [25]. Basically, the idea is to find the right mathematical model for the software system behavior and apply the corresponding treatment depending on whether the model is linear or non-linear. This direct application of the methods of classical process control might be tractable for simple cases, but most practical self-adaptive systems are too complex to apply these methods directly. The approach of [25] only considers the case where process control applies directly.

Here, we take a more nuanced position—we posit that understanding the basics of classical control theory provides a foundation for a more abstract view, based upon which important design guidelines, conceived especially for controlling software systems, can be generated and that might otherwise be overlooked. That is, we argue that control theory provides a point of view that enables a designer to design more complex self adaptive systems more effectively. The control-based method brings separation of concerns between, on the one hand, the design of the process or system to be controlled, and, on the other hand, the controller for a particular adaptation strategy (as distinguished from ad-hoc approaches where both are mixed, making them more difficult to design and reuse). It also brings assurances offered by the model-based approach, even if the software systems as object of control are quite different from the classical target of control theory.

This section follows [31] and introduces the concepts and vocabulary of control theory. In particular, it discusses factors that affect the quality of control that can be achieved. At each stage, it discusses how control algorithms affect the behavior of systems and it identifies more abstract design tasks that are extensions of the basic theory. We focus here on discretized control, that is, on models with discrete time steps.

In a simple feedback system, a process P has a tuning parameter u (e.g., a knob, which is represented by the little squared box in Fig. 2) that can be manipulated by a controller C to change the behavior of the process, and a tracked metric y that can be sensed in some way. The system is expected to maintain the tracked metric y at a reference level y_r , as illustrated in Fig. 2. At each time increment of the system, C compares the value of y (subject to possible signal translation) to the desired value y_r . The difference is the tracking or control error, and if they are sufficiently different, the controller changes parameter u to drive the process in such a way as to reduce the tracking error.¹ Somewhat confusingly, the tuning parameter u is often called the “input” and the tracked metric y is often called the “output.” It is important to remember that these are the “control” input and output, not the input and output of whatever computation P is performing.

¹ If appropriate, the process P can be described by a model and the controller C might use that model to determine changes of u .

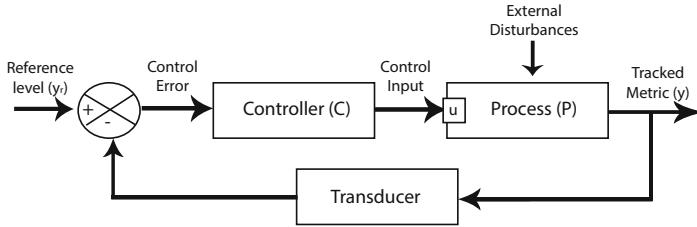


Fig. 2. The feedback loop control model.

For instance, assume the process P is a household thermostat to keep a room warm at a reference level of $y_r = 23^\circ\text{C}$, and the controller C is a rudimentary *on/off* decision mechanism. Correspondingly, the tuning parameter u in P is a simple switch *on/off*. The tracked metric y is the temperature measured from a strategically located sensor in the room. Regularly, C compares the current value of y to the desired value y_r . If y is less than y_r , C makes u to be in the *on* position. Of course, in a next cycle, whenever y is greater or equal than y_r , C makes u to turn *off*, and so on.

Control systems are especially important when processes are subject to unpredictable disturbances, illustrated in Fig. 2 as “External Disturbances.” External disturbances are inputs into the Process P that affect the controlled output and they cannot be controlled. For our household thermostat example, external disturbances are the external variations in temperature, together with the room temperature changes created by the sudden opening of the room door and windows. In computing environments, external disturbances examples include, among others, variable loads, such as number of users or request arrival rates, and variable hit rates on caches. Control systems can also be important when the computation in the process itself is unpredictable and its accuracy or performance can be tuned by adjusting its parameters.

Control systems can also concern events and state based aspects, where we have not only discretized time as before, but also a discrete state space, which can be infinite, or finite but non-trivial because of the size and/or transitions. There exist different approaches to discrete control in self-adaptive systems: some are related to planning techniques from Artificial Intelligence or reactive synthesis in Formal Methods or Game Theory. An approach stemming from the Control Theory community is the supervisory control of Discrete Event Systems (DES).

3.1 Correspondences Between Feedback Control and MAPE-K

An important difference between the feedback loop from control theory and the MAPE-K loop is the complexity of each constituent component and of the overall loop. For example, in the former, the control actions are atomic operations for physical actuators (*e.g.*, resistors and motors) with clear causality between inputs and outputs, whereas in the latter, they are sequences of discrete plans (thus called reconfiguration plans) with long lags and uncertain consequences.

However, the basic principles are the same and, in the SAS domain, it is useful to identify the analogues of the control elements, ensuring that they can be used effectively for control. For example:

- What corresponds to the tracked metric (or output)? That is, what characteristics of the process are we trying to control? Can we measure them directly? If not, how are we going to infer their values? For example, is there a proxy value readily available? How accurately does the proxy value reflect the true value?
- What corresponds to the reference level? That is, how do we characterize the desired behavior of the system? Is it a specific target? Is it a limit value, so that y should always be less than the limit (*e.g.*, the temperature should approach the boiling point in a cooking pot but never reach it), or are values on either side of the target acceptable (*e.g.*, the cooking pot temperature should be 75 °C)? Is it a function of something else? If it is a complex function of other system parameters, how do we analyze interactions between the control discipline and the inputs to the function?
- What corresponds to the tuning parameters of the process and the way of modifying them? That is, in what ways can the controller modify the behavior of the process? Are these ways sufficient to maintain control of the process? (Old-style automotive cruise control systems failed this test, because they only control the accelerator, so they could only speed up the car, not slow it down).
- What effect a given change will have in the control input have on the process? In control theory, this is called the system identification problem. The more precisely we know this effect, the better our ability to achieve the response we want. For complex SAS systems, a full specification is often impractical. If so, it is more important to know whether increasing u will increase or decrease y , than it is to know exactly what the size of the effect will be.
- What are the sources of the external disturbances? What is their nature and how do we characterize them? What is their frequency and amplitude? In control theory, feedback loops are designed also to compensate for the effect of disturbances, within some bounds in amplitude and frequency. Can we characterize those bounds for SAS?
- How does the controller decide how much to change the control input variables? Control theory offers a rich space of control disciplines with well-understood effects on the controlled system, as explained later in Sect. 6.1. What are the analogous theories for SAS systems?

Figure 3 depicts structural correspondences between the feedback loop and the MAPE-K loop (Autonomic Manager). The *reference level* (y_r) of desired properties is fed by the system administrator to the analysis phase of the MAPE-K loop. In this phase, the analyzer evaluates the difference between the *tracked metric* y and the reference level, which is captured by the sensors of the Autonomic Manager. The analysis phase communicates the adaptation decision to the planning phase by sending a change request that corresponds to the *control error*.

The planning and execution phases correspond to the *controller*, which generates an adaptation plan and executes the adaptation of the managed system, *the process*, through the effector using the *control input u* (*i.e.*, the adaptation commands). Finally, sensors may play the role of the *transducer*.

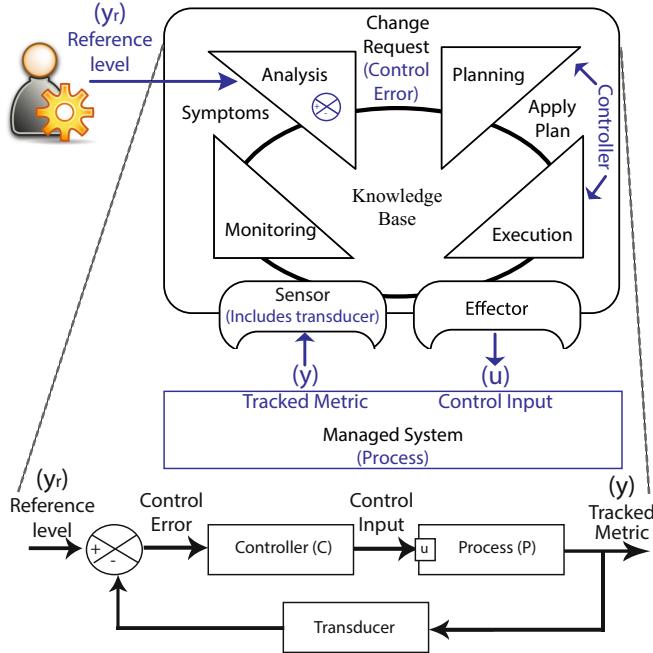


Fig. 3. Correspondences between the feedback loop and the MAPE-K Loop

The application of control theory to the engineering of self-adaptive applications has been actively studied by researchers in the SAS realm [12, 16, 38]. In particular, recent works have studied the usage of control theory to guide the design of MAPE-type systems, in different aspects of self-adaptation from the design of the adaptation mechanism to the assurance of its properties [56, 58, 60, 61, 63]. To continue advancing in this direction, an important step forward is to support reasoning about the systems by finding the answers to the questions listed above about the correspondences between the feedback loop model from control theory and the MAPE-K loop model from SAS. These questions apply to a wide range of SAS systems, not simply to low-level control systems.

4 Adaptive and Hierarchical Control

For systems that vary in time or face a wide range of external disturbances, it is basically impossible to design one controller that addresses all those changes.

For these cases, we need to design a set of controllers or a parameterized controller. When the current controller is not efficient anymore, we change it or retune it. When the controller is updated while the system runs, we call it adaptive control. Adaptive control requires additional logic that monitors the efficiency of the controller under given conditions and, when some conditions are met, retunes the controller to adapt it to the new situation. The control community has dealt with adaptive control for decades, since 1961 according to Åström [5]. Åström provides a good working definition for adaptive control: “An adaptive controller is a controller with adjustable parameters and a mechanism for adjusting the parameters.” This definition implies two (or more) layered control loops. Regular home thermostats are simple instances of on-off control that have no information about environment other than temperature. An hypothetical “adaptive thermostat” would use additional environment variables as well, such as humidity, door or window open. The values of these additional parameters (through some tuning parameters) would change the hypothetical thermostat control strategy (on-off switch times, for example). In the SAS area, the use of adaptive feedback loops was first suggested in [41]. One possible realization of an adaptive controller is depicted in Fig. 4. There are two feedback loops: the reactive feedback loop, at the bottom, that takes the output y and brings it to the comparator and further to the controller; and the adaptation loop at the top that estimates a dynamic set of models of the software (Model Estimators) passes the models (Models) to a Controller Tuning/Synthesizing component. This component uses a current Model to either recalculate the parameters of the controller or switch to a different controller.

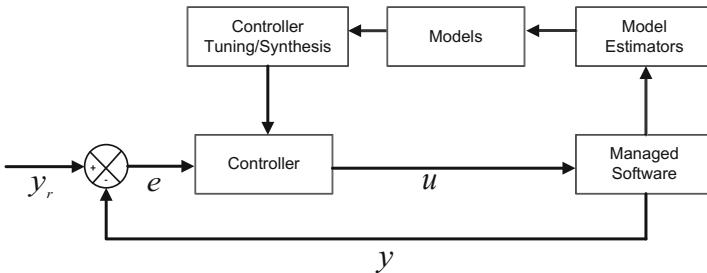


Fig. 4. Model Identification Adaptive Control

Hierarchical control, in addition to adaptive control, can be used for complex systems and complex controllers. The robotics community in particular has dealt with hierarchical control since the early eighties—they called it hierarchical intelligent control system (HICS). In the software engineering community, most notably Kephart and Chess [30,33] introduced autonomic systems and autonomic computing reference architecture (ACRA).

Concerning discrete control, notions of hierarchies have also been considered. One is the fact that there is often a need for coordination of, or switching

between, different quantitative controllers, each with specific objectives. Between levels of discrete control, hierarchies can also be established as in *e.g.*, [19, 21]. Adaptive discrete control, on the other hand, is a notion less developed than for quantitative control, and more theoretical work is needed.

The autonomic computing reference architecture (ACRA), depicted in Fig. 5, provides a reference architecture as a guide to organize and orchestrate SAS (*i.e.*, autonomic) systems using autonomic managers (MAPE-K loops). SAS systems based on ACRA are defined as a set of hierarchically structured controllers. Using ACRA, software policies and assurances can be implemented into layers where system administrator (manual manager) policies control lower level policies. System operators have access to all ACRA levels.

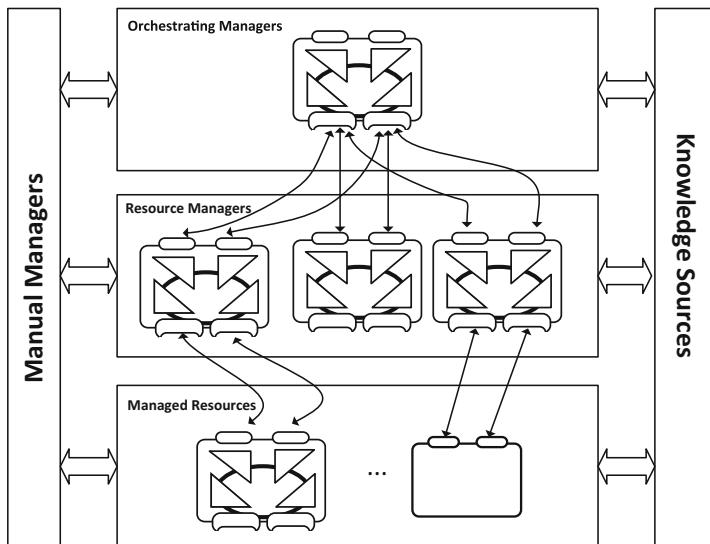


Fig. 5. The Autonomic Computing Reference Architecture (ACRA) [30]

Hierarchical control for SAS was further specialized by Kramer and Magee in a three layers architecture for SAS [34]. Each layer has a different time scope and authority. The lowest layer, *component control*, contains all the mechanisms to support changes in components; the middle layer, *change management*, contains the plans that are executed by the lowest layer; and the upper layer, *goal management*, produces change management plans as requested by the layer below and in response to the introduction of new goals.

5 Control Theory Applied to Self Adaptive Software - An Overview

Over the last decade, there have been many efforts to use control theory to design feedback loops for self-adaptive software systems. Hellerstein *et al.*, in their 2004

book [29], introduced several examples where control theory has been applied for controlling the threading level, memory allocation or buffer pool sharing in commercial products, such as IBM Lotus Notes and IBM DB2. Other researchers and practitioners have published results on control theory for computer power control [35], thread and web cluster management [1], admission control, video compression [24], performance and denial of service attack mitigation (*e.g.*, in the Apache server) [20], to name just a few examples.

In the examples we have seen so far, the authors use feedback loops to control quantitative metrics from the categories of performance, cost, energy, reliability. In these control examples, the methodology and the mathematical apparatus follow the control theory methodology and revolves initially around two basic concepts: the model of the controlled subsystem (*i.e.*, the open loop model from the point of view of the controlled metric), and the controller to close the loop, as follows.

5.1 The Open Loop Model

The first major task in applying control theory to SAS is to create a model for the quality attributes to be controlled. Since this model focuses only on the controlled or managed system (that is, the output is not considered to compute the control input), we call it open loop model. The model is quantitative and captures, in a very specific format, the relationships among software components, and between software and environment. The model can be constructed analytically by writing the equations of the modelled phenomena or by following an experimental methodology called system identification [39, 51]. In general, defining the model starts by identifying:

- the outputs that need to be controlled, y ;
- the disturbances, p , that affect the outputs;
- the control variables (or commands), u , that can control the outputs of the system and compensate for the effects of external disturbances; and
- the internal state variables, x (x can be seen as an intermediate link between u and y).

The external disturbances, p , drive the system towards undesired states. For example, an external load increase (increase in number of users) might increase the utilization (state) of a server and therefore increase the response time (output). Note that the perturbations p do not affect the response time directly, but through the intermediate state that we call utilization. To compensate the effects of the external load, a feedback loop designer should engineer a control input that changes the utilization of server in the opposite direction (like transferring some of the load to another server). Note that u , x , p , y are vectors, not scalars, which means that a system has many input, output, state or external disturbance variables. In control terminology, this is defined as MIMO (multiple input, multiple output) system.

In the general case, with f and g non-linear discrete *vector* functions, the open loop model can be described as:²

$$x(k+1) = f(x(k), u(k), p(k)) \quad (1)$$

$$y(k) = g(x(k), u(k)) \quad (2)$$

where k is the current discrete time and $k+1$ means the “next time increment.” The magnitude of the difference between k and $k+1$ depends on exactly what the model tries to capture. Equation 1 projects the state forward, as a function of the current state x , some control commands u and disturbances/perturbations, while Eq. 2 expresses the output as a function of the same parameters.

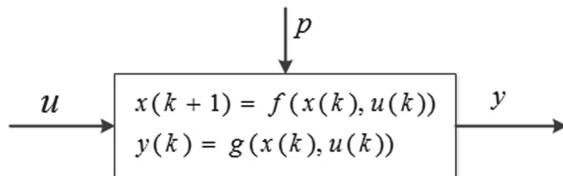


Fig. 6. A discrete non-linear model

The above model formulation, omitting the disturbance p as depicted in Fig. 6, is general enough to describe any quantitative aspect of a system that might need adaptation, including aspects of SAS. When constructing the above model for a software system, a special attention should be paid to identifying (or engineering) u , the control inputs or commands. Those are variables in the system that control engineers pay special attention to and they are a *sine qua non* condition for designing and engineering a feedback loop. The commands u are control knobs, that is software parameters, scripts, programs, interfaces through which a system administrator or an automation program can change a software configuration. They are always designed with some knowledge about the possible external disturbances. Load inputs to the systems, such as workloads, are frequently modeled as external disturbances or perturbations, p , because they are outside the control of the SAS designer. Their effects are compensated for by designing control inputs u , engineered into the system: for example by increasing the capacity of the server, scaling out by replicating servers and distributing the load, changing the data flows inside the system or changing the threading levels.

5.1.1 Continuous Linear Open Loop Models

In many cases, the non-linear model can be simplified or approximated with a linear one, such as the one below. In this model we assume an additive external perturbation:

² In many cases the models include explicitly the modelling and measurement noise; to keep the presentation simple, in this chapter we do not represent noise explicitly.

$$x(k+1) = A * x(k) + B * u(k) + F * p(k) \quad (3)$$

$$y(k) = C * x(k) + D * u(k) \quad (4)$$

where A, B, C, D, F are constant matrices that can be determined experimentally for a specific deployment and under particular perturbations.³

Example 1: Software queues modeling (performance view). Many software entities can be modelled as queues (semaphores, critical sections, thread and connection pool containers, web services, servers, etc.). Although the performance metrics of those entities can be captured with non-linear models, many authors prefer to linearize the models around an equilibrium point and express them with the canonical model (3) and (4). Below is an example of such model for a web server (taken from [20, 29]). In Eqs. (5) and (6), x_1 and x_2 represent the server utilization and the memory usage, respectively; the commands, u_1 and u_2 represent the number of HTTP connections and the keep-alive intervals; and y represents the response time of the server. A_{ij} , B_{ij} , where $i, j = 1, 2$ and C_1 are constants that have to be determined experimentally for a specific server deployment. Intuitively, the model says that the future memory usage and the server utilizations are a function of the current usage, the number of connections, and the length of interval they are kept alive.

$$\begin{bmatrix} x_1(k+1) \\ x_2(k+1) \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix} + \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \begin{bmatrix} u_1(k) \\ u_2(k) \end{bmatrix} \quad (5)$$

$$y(k) = [C_1 \ 0] \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix} \quad (6)$$

Similar models have been presented for various entities that contain queues [3, 24, 28, 52, 53]. They were derived from the dynamics of the queues and from experiments. For example, Solomon *et al.* [53] use a control theory specific model identification methodology to experimentally identify A, B, C , and D for a threading pool.

5.1.2 Discrete Open Loop Models

If in contrast a finite discrete model and discrete control is considered, we may use instead of a linear infinite model a discrete transition system. A simple case to describe the open loop model would be Moore automata $M = (X, \Sigma, \Omega, \delta, \mu)$ with state space X , input event set Σ , output event set Ω , total state transition function $\delta : \Sigma \times X \rightarrow X$, and total output function $\mu : X \rightarrow \Omega$. Assuming events sets U , P , and Y for u , p , and y , respectively, we could use a Moore automaton $M_{ss} = (X_{ss}, \Sigma_{ss}, \Omega_{ss}, \delta_{ss}, \mu_{ss}, x_{ss})$ with X_{ss} the possible states x , $\Sigma_{ss} = U \times P$, $\Omega_{ss} = Y$, and $x_{ss} \in X_{ss}$ the start state to describe the software system as follows:

$$x(k+1) = \delta_{ss}((u(k), p(k)), x(k)) \quad y(k) = \mu_{ss}(x(k)) \quad (7)$$

³ Here we assume a time invariant case, but those values can depend on time as well.

As we chose $\Sigma_{ss} = U \times P$, the automaton adjusts its state in a single step according to the command u and the disturbance p . Therefore, a simple scheme might be that $U = \{\text{pass}, \text{block}\}$, such that for *pass* the software system operates as usual and for *block* the software system ignores the external input.

5.2 The Closed Loop Model

In the general case the feedback control can only observe the software system represented by the open loop model at its output y , while the disturbance p and state x are not directly observable.

5.2.1 Continuous Linear Closed Loop Models

For the linear case, Eqs. (3) and (4) suggest that if we want to keep y at a predefined value y_r (the goal), we accomplish that by computing a command u_r based on p that compensates the disturbance (feed-forward control). In practice, that is impossible for two main reasons: (a) the model (2) is an inaccurate representation of the real system; (b) there are external disturbances, p , that affect the system. A feedback loop implies adding a new software component, a controller (or Adaptive Manager) as depicted in Fig. 7, that is fed back with information from the controlled software system.

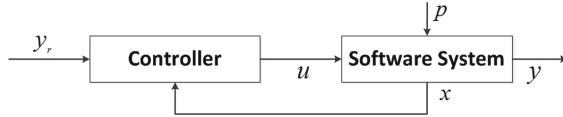


Fig. 7. Controller and feedback loop.

The controller and the feedback will compensate for modeling errors and for external disturbances. Furthermore, they can also stabilize unstable systems and achieve additional control criteria. The feedback can be taken from the state variables (state feedback) or from the output (output feedback). The state feedback is the most effective, since it assumes an understanding of inner workings and dependencies of the software system. Output feedback is more practical since output variables are easier to measure. For observable systems, a system for which its internal state variables can be estimated from measured variables, a state feedback can be constructed from output feedback, using estimators as for example Kalman filters [32]. When using estimators, we measure y ; estimate x based on the model, the observability matrix and the measurements; and then compute a state based controller. The controller can vary from a simple constant matrix to a complex set of differential (or difference) equations and can be synthesized to achieve some design goal, such as stability of the entire system, perturbations rejection across wide ranges (robust control), or optimization. While the controller design goals can be diverse, there are standard design techniques that have well defined procedures and solutions.

The interesting aspect of the feedback controller is that the overall equations of the closed loop system (or closed loop model) can be inferred from the open loop model and the equations of the controller. For example, consider the controller a matrix K , that needs to be determined. If K is placed on the feedback loop, then $u(k) = y_r \cdot K^* \cdot x(k)$. The equations of the closed loop system, having y_r as input and y as output, can be computed by substituting u in Eqs. (2) and (3), and will look as follows (for simplicity, $F = 0$):

$$x(k+1) = A_c * x(k) + B_c * y_r(k) \quad (8)$$

$$y(k) = C_c * x(k) + D_c * y_r(k) \quad (9)$$

where $A_c = (A - BK)$; $B_c = B$; $C_c = (C - DK)$; $D_c = D$.

Having the equations of the closed loop system allows the designer to shape the behavior of the states and therefore of the outputs by choosing the right K . At the same time, a model will enable the analysis and verification that the system has indeed the desired behavior.

5.2.2 Discrete Closed Loop Models

For the case of discrete control, a Moore automaton $M_c = (X_c, \Sigma_c, \Omega_c, \delta_c, \mu_c, x_c)$ with X_c the possible states of the controller, $\Sigma_c = Y$, $\Omega_c = U$, and $x_c \in X_c$ the start state to describe the controller for the software system of Eq. 7, the resulting closed loop model is as follows for x' the state component of the controller⁴ and x the state of the software system:

$$x'(k+1) = \delta_c(y(k), x'(k)) \quad (10)$$

$$x(k+1) = \delta_{ss}((\mu_c(x'(k)), p(k)), x(k)) \quad y(k) = \mu_{ss}(x(k)) \quad (11)$$

For the system described in Sect. 5.1.2, consider simple scheme with $U = \{pass, block\}$, such that for *pass* the software system operates as usual and for *block* the software system ignores the external input. A controller can thus always block an input if this would lead to an unsafe behavior. However, the controller has to deduce this from the observed outputs of the software system.

5.3 Feedback Control Behavior

The main challenge now is to chose the right strategy for the controller such that the closed loop model and its feedback control behavior has the required properties.

⁴ To stay closer to the continuous case, we consider here the case of a discrete controller with an own state space distinct from the state space of the controller process that is more general than the supervisory control approach.

5.3.1 Continuous Linear Control

In the case of continuous control, independent of having a model for the open or closed loop, a controller for a SAS achieves its goals by successive approximations. It is hard to know exactly how a given system will respond to changes in the control inputs (*i.e.*, the system dynamics).

Control algorithms operate by making successive corrections that are designed to reduce the tracking error. Several factors lead to this strategy, all of them derived from the uncertainties about the exact system dynamics: delays in system response (*e.g.*, startup time to bring a new server online), lags in system response (*e.g.*, the time it takes to clear a backlog), environmental changes and events (*e.g.*, the execution of the garbage collector), and variability in the interaction of the controlled process with its environment (*e.g.*, randomness in arrival rates to a queue, or processing times for the jobs in a queue). Feedback loops provide some robustness in the face of these uncertainties. However, the consequence of their use is that the process approaches the reference level in a series of steps. For example, in the case of the home thermostat, the controller turns the furnace *on* and evaluates how the temperature goes with respect to the reference value. Depending on the results, the thermostat controller turns the furnace *off* and evaluates the results again. This process is continuously repeated to reach the desired temperature.

The design of the control algorithm determines whether the path to achieving the reference value is a series of overshoots above and below the desired value or a gradual approach to the reference value. The former strategy may get close to the reference value faster, but at the cost of oscillating behavior (imagine riding in a car that entered a new speed zone and made drastic changes up and down in speed, first overshooting and then undershooting to get to the new speed limit). The latter strategy may take longer to reach the reference value, but with much smoother behavior and less resource wasting. These are the aspects of concern when analyzing control properties and respective assurances.

Example 2: Balancing Job Throughput and Parallelism in Hadoop. In recent years, Hadoop has emerged as the *de facto* standard for big data processing and the MapReduce paradigm has been applied to a wide variety of applications and workloads, including distributed sorting, log analysis, document clustering and machine learning just to name a few. In this context, the performance and the resource consumption of Hadoop jobs do not only depend on the characteristics of applications and workloads, but also on an appropriately configured Hadoop environment. Next to the infrastructure-level configuration (*e.g.*, the number of nodes in a cluster), the Hadoop performance is affected by job- and system-level parameter settings. For example, YARN (*Yet Another Resource Negotiator*), the resource manager introduced in the new generation of Hadoop (version 2.0), defines a number of parameters that control how the MapReduce jobs are scheduled in a cluster, affecting the global jobs performance. Among these parameters, the MARP (*Maximum Application Master Resource in Percent*) property directly affects the level of MapReduce job parallelism and associated throughput. This parameter controls how the capacity scheduler balances between the

number of concurrently executing MapReduce jobs and the number of map and reduce tasks. An inappropriate MARP configuration will therefore either reduce the number of jobs running in parallel resulting in idle jobs or reduce the number of map/reduce tasks and thus delaying the job completion. However, finding an appropriate MARP value is far from trivial. On the one hand, the diversity of MapReduce applications and workloads suggests that a simple, one-size-fits-all application-oblivious configuration will not be broadly effective—i.e., one MARP value that works well for one MapReduce application/workflow combination might not work for another. On the other hand, YARN configuration is static and as such it cannot reflect any changes in workload dynamics. In this context, we propose the design of a YARN controller that autonomously balances the throughput and the parallelism of Hadoop jobs in order to minimize their completion time and maximize the resource utilization of the Hadoop cluster. This control problem can therefore be modeled as follows (cf. Fig. 8).

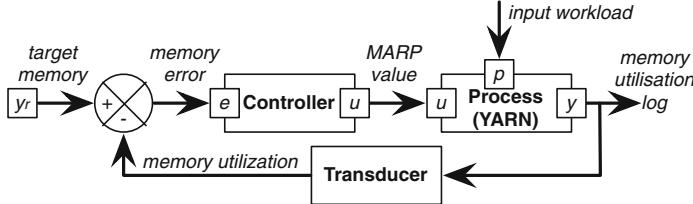


Fig. 8. Self-balancing controller for YARN.

To assess this controller, we executed all experiments on an Hadoop cluster with 11 physical hosts (1 control node and 10 processing nodes) deployed on the Grid5000 infrastructure.⁵ We use SWIM (*Statistical Workload Injector for Mapreduce*) to generate 4 realistic MapReduce heterogeneous workloads. SWIM contains several large workloads (thousands of jobs), with complex data, arrival, and computation patterns that were synthesized from historical traces from Facebook 600-nodes. Figure 9 illustrates that one can observe for each workload that, compared to the vanilla configuration, our approach can significantly reduce the completion time of jobs (e.g., up to 40% in W1). It also systematically delivers a better performance than the best-effort configurations.

5.3.2 Discrete Control

Discrete control focuses not on quantitative aspects of a system as shown before, but rather on logical, event-based abstractions. In addition to a discretized time coming with the notion of events, this form of control considers a discrete state space, which can be infinite, or finite but non-trivial because of its size and/or complexity of transitions.

⁵ <https://www.grid5000.fr>.

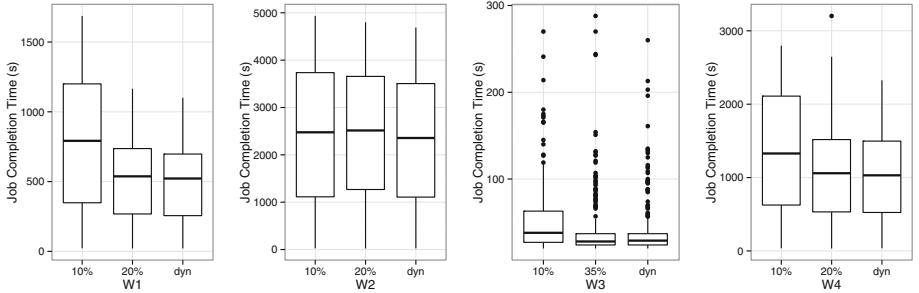


Fig. 9. The comparison of job absolute completion time observed for static and dynamic configuration parameters.

For the discrete control example with Moore automata $M_{ss} = (X_{ss}, \Sigma_{ss}, \Omega_{ss}, \delta_{ss}, \mu_{ss}, x_{ss})$ with X_{ss} the possible states x , $\Sigma_{ss} = U \times P$, $\Omega_{ss} = Y$, and $x_{ss} \in X_{ss}$ the start state to describe the software system and $M_c = (X_c, \Sigma_c, \Omega_c, \delta_c, \mu_c, x_c)$ with X_c the possible states of the controller, $\Sigma_c = Y$, $\Omega_c = U$, and $x_c \in X_c$ the start state to describe the controller for the software system combined in Eq. 10 with the simple blocking scheme supported by the control input u , the safety objective may be to restrict automata M_{ss} to the safe states $X_{ss}^{safe} \subset X_{ss}$. Then, the controller synthesis would synthesize a Moore automata M_c that blocks the Moore automata M_{ss} where necessary, such that for every sequence of external inputs p holds that M_{ss} never reaches an unsafe state in $X_{ss} \setminus X_{ss}^{safe}$. Maximally permissive here means that M_c use the *block* control input only to the absolute minimal extent that is necessary to exclude unsafe states.

There exist different approaches to discrete control in self-adaptive systems: some are related to planning techniques from artificial intelligence *e.g.*, [10, 21, 54], game theory or reactive synthesis in formal methods *e.g.*, [9, 22]. Such approaches concern control objectives as expressive as of safety, reachability or even liveness. An approach stemming from the control theory community is the supervisory control of Discrete Event Systems (DES) [14, 47, 62]. Typical models are transition systems like Petri nets or automata, and the properties considered are logical and pertain to states and paths in the state space, like safety, reachability and non-blocking. For some classes of problems, there exist automated state-space exploration algorithms for Discrete Controller Synthesis (DCS), some of them implemented in efficient tools. One particularity is that, for safety objectives, the controller can be ensured to be maximally permissive (see also [48]).

In the next section, based on the model, the controller and the control behavior definitions, we analyze different control strategies and the assurances they provide with respect to control properties, the analysis of the open model properties, the design (or synthesis) of the controller, and then the analysis of the resulting closed loop model.

6 Assurances

Most physical processes, and many computational processes (especially complex ones), do not respond immediately to changes in their control inputs. Further, most systems do not respond so precisely that an exact correction can be made in a single step. Therefore, the design of a feedback system, whether it be a simple feedback loop or a complex SAS, must take into account the way the controlled process responds to changes in the control inputs. The characteristic responses constitute the basis on which system properties are built, and the assurances correspond to what (and to what extent) can be guaranteed about these properties.

This section focuses on *assurances* that can be realized with the help of feedback loops, and we explore their correspondences in SAS domains. In other words, we focus on what we can ascertain when using feedback loops, and explore the correspondences of these assurances in SAS domains. Of course, ideally, it would be desirable that just by using feedback loops we would have granted assurances on desirable properties and optimum behavior in the controlled software applications. For instance, for regulating the throughput of the web service for computing definite integrals it would be desirable to have a controller that always (i) corrects errors quickly; (ii) reaches the reference level with precision and minimum error; (iii) spends the resources at the minimum required; and (iv) reaches stability after achieving the reference level. However, obtaining such a controller for a SAS system in general is not easy and constitutes precisely the main challenge. Moreover, conflicts often arise when trying to achieve all of these control objectives together (*e.g.*, fast settling time usually imply significant overshoot), and thus, trade-offs must be considered between these objectives and the implied assurances.

In the following subsections, we discuss different control strategies and the way they can be exploited to assure properties and desirable behaviors. Then, we analyze the open loop model properties, the conditions that can help design controllers that provide assurances on desirable properties, and then the analysis of the resulting closed loop model properties. Even though we do not provide a complete answer to the main challenge, this analysis helps understand better the implied problems and sets the basis for defining the principles on which a software controlling theory should emerge.

6.1 Classic Control Strategies

From the discussions of previous sections and the questions posed in Sect. 3.1, we can infer that decisions about the control strategy are absolutely critical to the design of SAS. Classical control theory operates in settings that are simple enough that mathematical analysis is usually possible. SAS systems, on the other

hand, are usually too complex to characterize completely. This makes the design and validation of the control strategy even more important for this kind of systems.⁶

This section reviews common control strategies and discusses their application to SAS. For this application, we use an extremely simplified version of the WolframAlpha web services⁷ as the software to be controlled. Our web service only computes the integral of elementary functions (*i.e.*, functions with exponentials, logarithms, radicals, trigonometric functions, and the four basic arithmetic operations).

The service is implemented using an exact method for computing definite integrals of a given elementary function, that is, by computing the symbolic integration of the function and evaluating the result on the integration limits. This method is slow (*i.e.*, worst case is exponential, even though for many cases it is polynomial [11]), but exact.

6.1.1 On/Off Control

The simplest type of control is for a system where the control inputs of the managed system are either turned *on* or *off*. The most common example is the old-style household thermostat, where the thermostat turns the furnace *on* or *off*, as described previously (cf. Sect. 3). This simple strategy has significant drawbacks: it oscillates constantly (and hence is often called “bang-bang” control). This problem can be moderated by introducing hysteresis, that is, by delaying the control response until a modest overshoot has occurred.

To illustrate the application of the *on/off* control to our web service for computing definite integrals, assume we deploy the control method implementation both on a main server and on a spare server. Assume also that the required reference level is to maintain a given throughput of definite integral computation requests per time unit. Whenever the main server can service the requests maintaining the required throughput, the controller makes u to turn *off* the spare server, such that the spare server does not accept any new requests and shuts down upon finishing any pending requests. Otherwise, the controller makes u to turn the spare server *on* and redirects requests to it. However, independent of whether the number of requests grows in greater magnitudes, the action of this type of controller is clearly limited to activating the spare server.

Similarly, for the hadoop example we can assume a predefined set of spare processing nodes⁸ that can be switched on and off depending on the requests throughput.

⁶ We often describe “self-adaptation” in terms of strategic changes to maintain the system behavior as close as possible to the reference value. Note that a system can also self-adapt by changing the control discipline (adaptive control). This is another reason to make explicit design decisions about the control strategy.

⁷ <http://www.wolframalpha.com>

⁸ Hadoop does not support the addition of processing nodes that have not been considered during initial deployment.

6.1.2 Proportional Control

This type of control acts by making the size of the adjustment on the control input proportional to the size of the tracking error. In the simple case, the constant of proportionality is called the gain. The proportional gain determines the ratio of output response to the tracking error. For instance, if the error term has a magnitude of 10, a proportional gain of 2 would produce a proportional response of 20. In general, increasing the proportional gain will increase the speed of the control system response. Even though this strategy means a fast response to eliminate the tracking error, a shortcoming of proportional control is that it generally is unable to eliminate the tracking error entirely, a phenomenon called proportional droop. Thus, one critical design task is gain estimation. Gains that are too high result in excessive oscillation, possibly never reaching convergence toward the reference level, while gains that are too low result in sluggish performance. The effect of larger and smaller gains is further illustrated in Sect. 6.5.3.

Revisiting the thermostat example, modern high-efficiency furnaces run at several power levels:⁹ the level 1 is energy-efficient but produces lower temperatures (*i.e.*, has lower gain), and the others gradually produce more heat at the expense of efficiency (*i.e.*, have higher gains). For an automated furnace with eleven internal power levels (0 being the Off position, 10 the maximum power) covering a range between 18°C and 27°C, the thermostat's ordinary operation uses the energy-efficient level in most cases, but at the expense of slower adjustment of temperatures in the living area. Assuming the thermostat has a proportional gain of 3, and the temperature error is of 3°C, the controller would produce a command control for selecting the ninth power level position in order to correct the temperature rapidly; notice that since this is almost the maximum power level, maintaining it for too long would lead to significant overshoots, and therefore, correspondingly large oscillations.

For the application of this type of control to our web service, assume we deploy the *implementing* method on the servers of a cluster computing infrastructure, and that the required reference level is the same throughput as the defined for the *On/Off* control example. In this case, we estimate the gain as 2/100 times the tracking error. That is, in short, and roughly speaking, the resulting command control on this strategy is $u(k+1) = \frac{2}{100} * e(k)$, being $e(k) = y(k) - y_r(k)$. Thus, if the error reaches a difference of 200 in the throughput, the controller response is to activate four spare servers and distribute the service requests among them with the purpose of maintaining the required throughput. Nonetheless, despite the fast corrective action, small tracking errors (*i.e.*, below 50 in this case) produce no correction (*i.e.*, evidencing proportional droop).

Similarly, for the hadoop cluster, we can implement the MARP parameter tuning with this type of control by adjusting the percentage of resource allocated to application master, according to the current memory consumption.

⁹ <https://www.ecobee.com>

6.1.3 Integral Control

As analyzed in the proportional control, independent small tracking errors can produce null corrective actions. However, the accumulation of these errors sooner or later should produce a corrective action, possibly small, but significant enough. The accumulated errors are essentially the integral of the error (or sum for discrete systems), so this is called integral control. However, given that this strategy applied alone naturally results in very slow responses, a common control strategy is to combine an integral control term with a proportional one. This combination provides a fast response action, with elimination of remnant small tracking errors.

For the thermostat example, assume the reference level is set to $25^{\circ}C$, the controller having an integral coefficient of $3/4$, and the current tracked metric being $24.5^{\circ}C$. Having an integral term alone would yield a command control of $0.5 * 3/4 = 0.375$, that is, selecting the *Off* position for the furnace. After 3 more cycles with the same error (*i.e.*, with null corrective actions) a first corrective action is produced by setting the furnace on the first position ($0.375 * 3 = 1.125$). However, as this power level setting is too low to achieve and maintain $25^{\circ}C$, the controller will have to keep accumulating errors until reaching the eighth power level, in which, after some more cycles, it will stabilize (recall from the initial specifications that the tenth power level achieves $27^{\circ}C$). Once achieved the reference level, the temperature error will be diminished to near zero, contributing in this amount to the integral term. Indeed, the main contribution of the integral term, besides eliminating remnant tracking errors, is to maintain achieved levels of control (*i.e.*, the eighth power level) at a given point, in which other control terms (*e.g.*, the proportional one) would be null. Thus, this term models inertial load effects that are inherent in physical systems.

For the application of the integral control to our web service and hadoop case studies, assume the same deployment and required reference level as in the corresponding proportional control example. Also, let us define the integral control term as $I(k + 1) = I(k) + \frac{1}{5} * e(k)$, that is, $u(k + 1) = I(k + 1) = \sum_{j=1}^k \frac{1}{5} * e(j)$. Thus, an independent small tracking error $e(k)$ of 2 units produces no immediate corrective actions. Nonetheless, if this error is maintained, after three control cycles the accumulated error is of 6 units, and the integral control term reaches $6/5$. Then, the controller response is to activate one ($6/5 = 1.2$) spare server and distribute the service requests.

6.1.4 Derivative Control

It is important in many cases to predict or anticipate the software behavior trends. If we would know that in the near future the tracking error is going to increase, then we would take a proactive action now. In many cases, performing the command u , which in many cases is a work-flow, might take several minutes. This is called an execution lag. In these cases, the command will affect the output y not instantaneously, but with a lag of several minutes. It is better in these situations to anticipate the *trend*. The simplest way to accomplish it is

to determine the sign and magnitude of the error derivative (or difference for discrete systems, $e(k) - e(k-1)$), and calculate u as a function of this difference.

Applied to the thermostat example, assume the same reference level of $25^\circ C$, the controller having a derivative coefficient of $3/4$, and the current tracked metric being $24.5^\circ C$. This derivative controller will not react until the error *trend* becomes significant, given that the derivative term alone yields a command control of $(e(k) - e(k-1)) * 3/4$. When the error difference reaches $4/3$, the controller produces the first corrective action by setting the furnace in the first position ($(4/3) * (3/4) = 1$), which, depending on the control cycle duration, will correct the error trend the more or the less. Even though this power level can eliminate the error *trend*, in absence of other control terms it is too low to achieve and maintain $25^\circ C$. Moreover, as this controller reacts to error differences, trying to nullify the error trend, it alone will take a long time to reach the reference level set.

For the application of this type of control to our web service, assume the same deployment and required reference level as in the corresponding proportional control example. Also, let us define the derivative control command as $u(k+1) = \frac{1}{100} * (e(k) - e(k-1))$. Notice that, given that this strategy is based on the error trend, it should be used to complement other strategy: even if the error is large, but the trend is negative (e.g., $e(k) - e(k-1) = -100$) indicating that the error is diminishing, the controller response is to deactivate one spare server. However, if we combine it with the proportional strategy in our example, the command control would be $u(k+1) = \frac{2}{100} * e(k) + \frac{1}{100} * (e(k) - e(k-1))$. In this case, if the error presents a difference of 200 in the throughput, but the error difference trend is -100 , the corrective action is to activate 3 spare servers (not 4, as the proportional control alone would yield in this case).

Example 3: A PID Controller for Cluster Utilization. Figure 10 shows a PID controller presented by Gergin *et al.* [26] for the control of a software cluster. The controlled variable is the cluster utilization, y , and the command, u , is the number of software replicas within the cluster. A PID controller computes the number of replicas u as a function of error $e = y - y_r$, where y_r is the setpoint utilization for the cluster. The PID controller amplifies the error (K_p term), integrates all past errors (K_i term) and anticipate the direction of the error (K_d term). The coefficients K_p, K_i, K_d are determined experimentally or based on the system model in such a way that some quality control metrics (overshoot, rising and settling time, steady errors) are achieved.

6.1.5 Discrete Control

When the properties or characteristics which are to be controlled in a self-adaptive system concern logical or qualitative aspects rather than quantitative aspects, then a different type of control techniques has to be applied. Discrete control exists under different forms, for example planning techniques from Artificial Intelligence have been used to synthesize the sequences of low-level actions necessary to reconfigure a system according to high-level goals, or in more ambitious multi-level

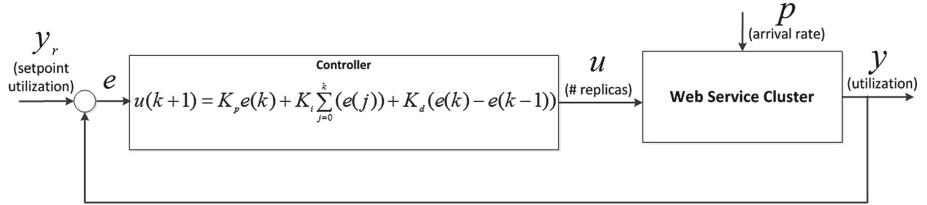


Fig. 10. A feedback loop with a PID controller for distributing the web service load in a cluster.

architectures to cope with unpredicted situations [21]. A form of discrete control, usually applied to flexible manufacturing, is the supervisory control of Discrete Event Systems (DES) [14]: their characterization is given by the nature of the state space of the considered system: when it can be described by a set of discrete values, like integers, or vectors of booleans, and state changes are observed only at discrete points in time, then such transitions between states are associated with events. These supervisory control techniques are beginning to be used for self-adaptive computing systems [48, 62].

6.2 Control Theory to the Rescue of the MAPE-K Loop

From the above discussion, it is apparent that each control strategy poses different advantages and disadvantages (*i.e.*, implying different properties). Moreover, some of the strategies can be combined, as in Example 2, being it usual to search for combinations that exploit the advantages of ones to compensate the disadvantages of others, given a particular control problem.

In addition to the ones discussed, there are many other control disciplines and strategies. However, the presented examples should be sufficient to show how critical it is to design the control strategy of a SAS system very carefully. It is important to notice that no other software engineering discipline forces attention to these matters. For instance, the well-known MAPE-K model, proposed for implementing autonomic computing and self-management computing systems, does not forcefully raise these design issues. Of course, the design of the planning element of the model should address these matters, but its specification is purely functional, being concerned fundamentally with *what* is required in order to modify the system behavior. Our discussion argues for balancing the concerns, considering even more importantly the longer term effects of the proposed behavior modifications, that is, the *properties* implied by the chosen control strategy, over the specific technique or method for producing the modification.

Any discussion about the properties of the closed loop system should start with a discussion about the properties of the open loop system. Only certain properties of the open loop system will allow us to design and analyze the closed loop properties.

In this setting, it is important to notice that a properly achieved property effectively becomes an assurance for the desired system behavior. Even

though the MAPE-K loop—and its further refinements—can be considered as an important contribution for SAS engineering, we strongly believe that the lessons learned by control theory in the assurance of desired properties is the direction we need to follow in order to consolidate it as a milestone. The remainder of this section makes the case for this claim by showing how control considerations provide a basis for reasoning about the control properties of systems.

6.3 Properties of the Open Loop Model

One of the most important reasons for having a model is to study the properties of the system it models. In the open loop model, the most important properties are stability, observability and controllability. Concerning assurance, we have to upfront understand their impact and implications to design the system such that required assurance can be obtained at all.

6.3.1 Stability

In simple terms, stability means that for bounded inputs (commands or perturbations), the system will produce bounded state and output values. In the case of Example 1 (cf. Sect. 5.1.1), a stable system might mean that for keep-alive connection intervals of 10 min (u_2) and for a 100 connections (u_1), the response time of the server is guaranteed to be between 1 and 2 s. Formally, stability is proven by necessary and sufficient conditions and, given a model, one can use Bode and Nyquist plots [4] to study stability. Examples of stability studies in control of software and computing systems have been presented in [3, 29]. Unfortunately, perturbations are the enemy of stability in open loop SAS, because the system is not set up to recognize how much the perturbations affect the system. If the open SAS is not stable, it can be stabilized through the design of a suitable controller. However, by analyzing the stability of the open system, we understand the source of instability and design the controller appropriately. For example, it is known that a source of instability is a saturated queue. It can lead to performance instability, faults and crashes. For a complex SAS, we have to assume which queue has the potential to be saturated by external disturbances, and then we design a controller that computes u such that avoids saturation.

For discrete open loop models, in contrast to continuous open loop models, often no metrics for the output y and external disturbance p exists and thus the concept of stability is not applicable.

6.3.2 Observability

Observability is the property of the model that allows to find, or at least estimate, the internal state variables of a system from the output variables. This property is important from a pragmatic point of view. In a real system, it is impossible, hard or impractical to measure all state variables. On the other hand, the commands and outputs are easier to measure. By knowing the model of the system in the form of Eqs. (3) and (4), if the matrices A and C are well behaved, one can compute x as a function of y . Formally, a model is observable if the observability matrix

$$O = [C \ C A \ C A^2, \dots, C A^{n-1}] \quad (12)$$

has the rank n , where n is the number of the state variables in x . Matrix O has to be invertible (or have rank n) in order to compute x as a function of y . Its structure is determined by writing the Eq. (4) for n values of k and considering $D=0$. Simple algebraic operations will allow us to compute x only if the matrix O is invertible. Examples of web service observability from a performance point of view can be found in [15]. Examples of how to estimate software performance parameters for applications deployed across multi-tiers and using Kalman filters are presented in [65].

The concept of observability can be transferred to discrete open loop models by considering whether the observable output y provides enough information to determine the state x . However, unless the start state is not known when the controller starts, the typical criteria employed is controllability in the sense that the existence of a controller to achieve the control objective is directly considered.

6.3.3 Controllability (or Reachability)

The concept of controllability (or state controllability) describes the possibility of driving the open system to a desired state, that is, to bring its internal state variables to certain values [15]. Of course, the system can be driven to a certain state by changing the command variables u . In the case of Example 1 (cf. Sect. 5.1.1), we should be able to find the values of u_1 and u_2 that will bring the server utilization (x_1) and memory usage (x_2) to 50%, for example. Like observability, the formal proof is given by the structure of the matrices that make the Eqs. (3) and (4): the system is controllable if the controllability matrix,

$$S = [B \ AB \ A^2B, \dots, A^{n-1}B] \quad (13)$$

has the rank n , where n is the number of state variables. The structure of matrix S is determined by writing the Eq. (4) for n values of k . Simple algebraic operations will lead us to matrix S and we can compute x as function of u only if S is invertible, hence has the rank n . If observability is not a necessary condition for designing a controller for SAS, the Controllability property is. Even we do not have an explicit open loop model, a qualitative analysis should be done. If the external disturbances, their frequency and amplitudes are known, do we have enough control inputs to compensate those perturbations?

Actually, the typical criteria employed for discrete control is controllability in the sense that the existence of a controller to achieve the control objective is directly considered. However, this perspective often differs from controllability in the above introduced sense. E.g., the earlier outline objective to stay within the set of safe states considered for controller synthesis does not equal controllability, which would only demand that the control input allows that any state can be reached, but not that the open loop model can then be forced to stay with a particular set of states.

6.4 Complex Open SAS and Model Composition

It is always the case that software systems are made of many components, interconnected together. The components might have different life cycles and be under different administrative domains. It is therefore imperative to answer the question: What are the properties of the overall open system (controllability, observability, stability) if we know the properties of the individual components? If the components are described by models such as the ones presented in Eqs. (3) and (4), then we have a method to answer that question. Models such as (3) and (4) can be composed, that is, if they are connected in series, parallel or through a feedback loop, a resultant composed model can be derived from the individual models. The composed model will have the same structure as the one presented in Eqs. (3) and (4), but the matrices A, B, C, D will be different but determined analytically from the individual models. Two services connected in series means that the output of the first one is the input of the second one (inputs and outputs have the control theoretic meaning). Parallel composition means that the same input reaches two different models and the output of the models will be summed up and used (eventually) as an input in other model. Feedback composition refers to a composition in which the output of a model is brought back and subtracted from the input of another model, as illustrated in Eqs. (7) and (8).

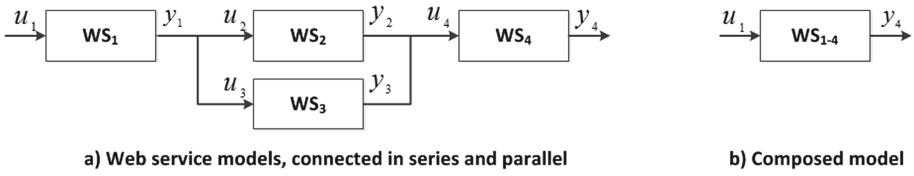


Fig. 11. Model composition for web services.

Example 3: Web service composition. In [52], the authors showed, based on web service performance case studies, how web services models can be composed in series and in parallel. Figure 11 illustrates the composition process: a set of web services are interconnected in an workflow; if we consider their performance, then the inputs (commands), u , are arrival rates and the outputs, y , are throughputs. For services in series, the throughput of the service on the left becomes the arrival rate for the service on the right. In Fig. 11a, WS_2 and WS_3 are connected in parallel, which means $u_2 = u_3 = y_1/2$ and $u_4 = y_2 + y_3$. By composing the models using their property of linearity, we can get a model of the entire system that is similar to the one presented in Eqs. (3) and (4). It is interesting to note that by composing web service models, the properties of the individual models (stability, controllability, observability) are not necessarily transferred to the composed model [15]. Model composition through feedback loops was briefly described by Eqs. (7) and (8).

6.4.1 Discrete Control

Also in case of discrete finite state models hold that composition would be necessary if larger systems should be considered. Furthermore, also in this case the composition usually does not preserve often desirable properties. Even worse oftentimes the composition can itself already lead to unwanted phenomena such as deadlocks. It is also interesting to note that for models that result from the composition of multiple transition systems oftentimes the time until which a control input may have impact on the state or output in the envisioned manner may become quite large, as each internal interaction between the composed transition systems leads to delays that sum up. In addition, the composition leads quite fast to very large state spaces as the orthogonal combination of the states of the composed transition systems has to be considered.

6.5 Properties of the Closed Loop Model

Once an open SAS has been analyzed and the designer has a good understanding of its stability, observability and controllability, a controller can be designed. When an explicit model of the open loop is available, the closed loop model can be synthesized mathematically to achieve the properties the designer wants. As Eqs. 7 and 8, suggest, A_c can be shaped and therefore the properties of the closed loop model. In general, the controller is designed to achieve some goals. The most frequent ones are described below. If assurance needs fit to one of these types of goals, control concepts can be employed to obtain the required assurance.

6.5.1 Stability

It refers to whether control corrections move the open system state, over time, toward the reference value. A system is unstable if the control causes overcorrections that never decrease, or that increase without limit. Instability can be introduced by making corrections that are too large in an attempt to achieve the reference level quickly. This leads to oscillating behaviors in which the system overshoots the reference value alternately to the high side and the low side. The opposite problem from stability is sluggish performance: if the control corrections are too small, it may take a very long time for the system to reach the reference value. In designing a control algorithm, the goal is to make the largest control correction that does not make the system unstable. Of course, the more we know about the way the system responds to changes in the tuning parameters (*i.e.*, the dynamics of the system), the easier it is to accomplish this goal.

If a model of the open loop exists, Eqs. (3) and (4), a controller can be designed to correct any instability of the open system that might be caused by perturbations or intrinsic properties of the software. The stability property is defined for the entire composed system, (considering y_r or p as input and y as output). With a model available, the stability is prescribed in the frequency domain (Z or S transforms) and implies solving a set of algebraic equations that will give the desired controller equations. This has been recently shown for

several SAS use cases. For example, a controller design for stability is presented by Cortellesa *et al.* [3] for a performance case study.

As in the case of the open loop model, for discrete closed loop models, in contrast to continuous closed loop models, often no metrics for the output y and external disturbance p exist, and thus the concept of stability is inapplicable.

6.5.2 Robustness or Robust Stability

A special type of stability in control theory is the *robust stability*. Robust stability means that the closed loop system is stable in the presence of external disturbances, model parameters and model structure uncertainties. If we refer to models (3) and (4), external disturbances uncertainties refer to wide variations of p in amplitude and frequency. Consider an application in which the disturbance is the external load, that is the number of users and the frequency of user requests. The number of users can vary from 100 to 1 million and their requests frequency from 0.1 to 40 requests per second. Can we design one single controller that maintain the performance of the system (response time) below a certain threshold in the presence of those large variations? Model parameter uncertainties in Eqs. (3) and (4) refer to errors in identifying A, B, C, D, F . How can we design a controller that can achieve stability when there are large errors in those parameters? Model structure uncertainties refer to working with models (3) and (4) that miss dynamics of the system (for example ignoring important state variables). Can the controller provide stability in this situation? In general, can we design one static controller that assure stability within quantifiable uncertainties bounds?

In control theory, there are several design techniques that help a designer achieve robust stability (*e.g.*, H_∞ , loop shaping or quantitative feedback theory or QFT). When a SAS can be described by a system of Eqs. (1) and (2) or (3) and (4), then we should consider those control techniques. For more complex SAS, it is probably feasible to consider Adaptive and/or Hierarchical Control where multiple controllers are synthesized or tuned dynamically to face wide disturbances.

For discrete models, in contrast to continuous linear models, often a single event that is different is sufficient to completely change the subsequent observable behavior. Consequently, robustness is in general difficult to achieve for these kind of models. Therefore, even if metrics for the output y and external disturbance p exists, robust stability is often not reasonable to expect in case of discrete models.

6.5.3 Performance

In addition to stability, a controller has to achieve certain performance metrics: rise time, overshoot, settling time, steady error or accuracy [4,8], as illustrated in our web service for computing definite integrals. For a system subject to a PID controller, such as the one treated in that example, all of these metrics depend on the selected values for coefficients K_p, K_i, K_d . Figure 12 indicates these metrics,

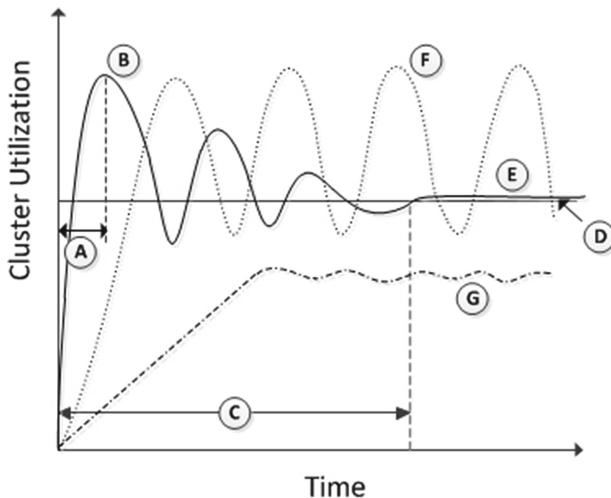


Fig. 12. Quality metrics for a controlled system [8]: rise time (A); overshoot (B); settling time (C); steady error or accuracy (D). Fine-tuned values for a PID controller, K_p , K_i , K_d , result in a smooth behavior reaching the reference level with accuracy and acceptable settling time (curve labeled (E)). Large values for K_p (and in lesser extents K_i and K_d), produce corresponding large oscillations around the reference level (curve labeled (F)). Small values usually result in a sluggish behavior (curve labeled (G)).

and illustrates the cluster utilization of a web service when perturbed by a workload change.

Assume the reference level is the horizontal continuous line in the figure. The perturbation, that affects the system at the origin of the time axis, drops the utilization to close to zero. A good controller (PID in this case) will compute a command u (the number of replicas in the cluster) that will bring the web service behavior back to normal in a short time (rise time) by varying the number of replicas in the cluster. Very likely, the system will be under- or over-provisioned, but the over- and undershooting must be kept small since it can affect other attributes, like response time and stability. Naturally, a few oscillations around the reference level are acceptable, but after a time (settling time), preferably short, the cluster utilization must reach a stable behavior (cf. curve labeled (E) in the figure). The figure also presents two examples of unstable and unacceptable control behavior: one which keeps oscillating around the reference level, and one which never reaches it (cf. curves labeled (F) and (G), respectively).

For discrete models, in contrast to continuous models, we often have no metrics for the output y , and thus, many of the above introduced terms cannot be employed. However, there are cases such as the synthesis of strategies for games, where the worst-case number of steps required to win the game could be seen as a criteria for performance.

6.5.4 Optimality or Linear Quadratic Regulator (LQR)

In many cases a controller has many goals, especially when there are multiple reference points, multiple input variables and multiple outputs. For example, a cruise controller should track the set speed but also minimize the consume of fuel. Usually the goals are conflicting. In control theory, the common way to treat this is to define an objective function across different goals. The goals can have different weights. Most commonly, the goals are defined across the state, output and control variables in the form of a quadratic function. The role of the controller is to find the commands u that minimize this quadratic function.

This type of controller with multiple goals is especially important for SAS systems, especially when the goal of the controller is a combination of conflicting goals, such as maintaining performance within certain bounds, optimizing the use of resources for saving cost and energy, and maximizing user experience. For example, Ghanbari *et al.* [28], for a SAS described by equations similar to (3) and (4), defined a quadratic function over the commands u (number of service instances) and the deviations from the setpoint y_r (or SLA violations) for cost and performance optimization in cloud computing, with Q and R matrices that penalize some elements of y and u :

$$J = \sum_{t=0}^k (y - y_r)^T Q (y - y_r) + u^T R u \quad (14)$$

They showed that a controller can be designed to optimize the sum of two conflicting goals (performance and cost).

In control theory, there are algorithms that assure the optimality (or sub-optimality) of the solution and those can applied to SAS that can be described by Eqs. (3) and (4). However, the definition of the objective function, practical ways to establish the weights of the goals, are subject of further studies.

6.5.5 Look-Ahead Optimality or Model Predictive Control (MPC)

In control theory, MPC optimizes a set of goals over a future time horizon H . This makes sense for economic goals, where variables such as the revenue or profit are calculated over a time horizon. The idea behind these controllers is to make strategic decisions, that is to make a current decision by taking into account the long term goals. In other words, a decision may be suboptimal now but may prove optimal in long term. The goal of the controller in this case is to find $u(k), u(k+1), \dots, u(k+H)$, with k the current time that optimizes a function similar to LQR over the future horizon H .

This type of optimization is especially important to complex SAS that have multiple design goals and have to prove efficiency over long periods of time. Examples of such optimizations, mostly in server provisioning for performance, cost and power can be found in [27, 35, 64]. The optimization problem for MPC problem looks like:

$$J = \sum_{t=k}^{k+H} (y - y_r)^T Q (y - y_r) + u^T R_u \quad (15)$$

In essence, a controller that optimizes J performs the following: if the current time is k , then the controller computes the future states x and the future outputs y for the next H time intervals. At the same time, the controller computes the controls u for these future H time intervals such that the entire sum in J equation is optimized. That is possible given the Eqs. (3) and (4). Pragmatically, the controller only implements the control $u(k)$ from the entire set of controls. At time $k+1$, the controller recomputes the commands for $k+1, \dots, k+H+1$ and, again, implements $u(k+1)$.

Optimality in discrete models can be defined in contrast based on the current state. If accumulated over the sequence of states the software system is in, we get an overall reward, which can be used as an optimization criteria for the synthesis of controllers. However, in case of non-determinism in the model (unknown effects due to p), the optimization can only take, for example, the worst or best case into account. Therefore, oftentimes probabilistic models are considered to then optimize the expected reward.

6.5.6 Additional Properties only Related to Discrete Control

The corresponding properties that are best managed by discrete control techniques have to do with subsets of a state space, characterized by predicates, or paths in the state space described by allowed and forbidden sequences of transitions. Such properties can be checked on a given transition system, representing the behavior of a program or system, *e.g.*, using model checking algorithms [17] for which there exist numerous tools. Some can also be submitted to synthesis, automatically producing an implementation satisfying the property *e.g.*, [9]. Such properties can be specified using formulae in temporal logics (*e.g.*, LTL, Linear Time Logic). *Safety* properties concern characteristics that should always, or never, occur, and typically concern reachability (or not) of some state (or states set, defined by a predicate), as for example two tasks accessing the same resource. They can also concern path properties, involving sequences of observations or of actions, such as elevator doors opening should never occur between starting and stopping of the vertical movement. Such safety properties can be the object of synthesis in the framework of supervisory control for DES [14], in an approach where the synthesized controller constrains the values of controllable variables so that, for all sequences of uncontrollable inputs values, the paths will keep satisfying the property (for examples, see [48]). Another type of property, for more expressive logics, concerns *liveness*, where something can eventually become true.

6.6 Open Questions

Control theory models provide fundamental ideas that can be used to leverage assurances in SAS adaptation mechanisms. The open and closed loop properties,

even if not captured in formalized models for SAS, must be considered by SAS designers along the system's engineering life cycle. Villegas *et al.* comprehensively define these properties in the context of SAS systems [58]. Implied questions are, for example:

- Above all, how can we determine whether a given SAS will be stable?
- How quickly will the system respond to a change in the reference value? Is this fast enough for the application? Control of a supersonic aircraft requires much faster response times than control of a slow-moving ground vehicle. Are there lags or delays that will affect the response time? If so, are they intrinsic to the system or can they be optimized?
- Are there some bounds of the external disturbances we know and how to design the SAS for?
- Can we design a robust controller to achieve robust stability or use Adaptive or Hierarchical Control?
- How accurately shall the system track the reference value? Is this good enough for the application?
- How much resources will the system spend in tracking the reference value? Can this amount be optimized? What is more important, minimizing the cost of resources or tracking the reference values? Can this tradeoff be quantified?
- It is very likely that multiple control inputs are needed to achieve robust stability. How will the control algorithm select the corrections to the control inputs?

7 Assurance Challenges for Self-Adaptive Software

From the examples presented in the previous sections, it should be clear that the concepts and principles of control theory and the assurances they provide, at least in abstract form, can be applied to a large class of problems in SAS. In the realm of SAS systems, we must consider at least two scenarios.

The first, in which it is possible to apply control theory *directly* to SAS, that is, by building a mathematical model (*e.g.*, a set of Eqs. (1) and (2) or automata) for the software system behavior, and applying control theory techniques to obtain properties, such as stability, performance, robustness (or robust stability), or safety and liveness properties. These properties automatically grant corresponding assurances about the controlled system behavior. This scenario requires two main necessary conditions in case of continuous linear models: the problem must refer to quantitative metrics, and it must be simple enough to be modeled as a linear set of equations relating control inputs, outputs and state variables in the open and controlled system.

In the case of discrete models, the scenario requires also two conditions: the problem must be of a logical nature, involving conditions and synchronization, such that it can be formulated as a controller synthesis problem, and must be simple enough to be modeled as a not too-large transition system. Nonetheless, there are still no clear guidelines about the limitations of control theory as directly applied to SAS systems in the general case.

The second scenario arises when it is infeasible to build a reasonably precise mathematical model, but instead, it is possible to create an approximated operational or even qualitative model of the SAS behavior. In this case, the formal definitions and techniques of control theory may not apply directly, but understanding the principles of control theory can guide the sorts of questions the designer should answer and take care of. In other words, even if a mathematical model is unavailable, control theory provides validation obligations and elements that can be exploited to model the phenomena to be controlled.

In this section, we highlight the main challenges a designer faces when dealing with the problem of designing SAS systems, and extrapolate the questions and concepts from the control theory approach into the general case of SAS.

7.1 Modeling Challenges

In both of the aforementioned scenarios, and based on the specification of the system properties to achieve, the SAS system designer must tackle the following main modeling challenges:

- *How to implement and instrument the control commands u in the software system to be controlled?* For instance, Hellerstein *et al.* control the number of active threads in the Apache thread pool by varying the respective parameter [29]. Even though this can appear as natural, it is unclear whether the Apache server was intentionally designed with this parameter to be modified automatically or rather to be configured manually at installation time. For any other given software system to be controlled, what parameter should be defined, and affecting what components and functionalities? Is the provision of one (or several) parameters the best alternative for implementing the control commands? Other options, very different to varying numerical parameters, are modifying the software structure in critical parts (*e.g.*, using domain-specific design patterns), as in [55]; or using machine-learning strategies to modify the system behavior, as in [23].
- *How to obtain an operational understanding of the effects* that control commands will have on the controlled system? That is, a variation of one unit on the control parameter affects in what magnitude the number of related components and the behavior of their functionalities?
- Given a set of control commands and the operational understanding of their effects on the system behavior, how to refine these commands to refine also the granularity of their effects, in order to improve the accuracy controllability?
- How many cycles of control are needed to reach the reference level fast? How to determine the change size to be applied in each control cycle to minimize overshoot?

7.1.1 The Core Phenomena to Control

In control theory, differential (or difference) equations are the models on which principles and properties of the phenomena to control are described and analyzed.

Depending on the behavioral characteristics of the target system, a controller can be defined to make the system behave as desired. The desired behavior is usually specified by either providing a reference input the system should follow (*e.g.*, PID controllers), or as an optimization problem (*e.g.*, Model Predictive Control) [5, 43]. The characteristics of a controller are usually defined by adjusting its parameters, which have special significance to generate the signals that will adapt the system depending on how far measured outputs are from the corresponding reference inputs.

Indeed, controllers are designed as parametric functions that generate the control signals that drive the target system towards accomplishing its goals. In software systems the identification of the core phenomena to control is typically a complex task. In contrast to physical systems, software systems still lack general methods to model the multi-dimensional and non-linear relationships between system goals and adaptation mechanisms [46, 58]; moreover, the problem and solution spaces can differ drastically (*e.g.*, in the web service example for computing integrals of elementary functions, calculating $\int \frac{x}{\sqrt{x^4+10x^2-96x-71}} dx$ takes polynomial time, but only changing the constant 71 to 72, that is, $\int \frac{x}{\sqrt{x^4+10x^2-96x-72}} dx$, takes exponential time, as analyzed in [11]). This simple example illustrates how complex the problem space can be in SAS systems, on which tracked metrics depend upon, directly and explicitly.

Furthermore, an adaptation mechanism can reconfigure the software structure by applying domain-specific design patterns with the goal of improving the system performance, as realized in [55]. Examples of these design patterns and strategies include Leaders/Followers, Half Sync/Half Async, Load Balancer, and Master/Worker, among others [2, 7, 13, 40]. However, it is still an open challenge to model the exact effect of the structural and behavioral elements introduced by a pattern in the system performance.

Assurance Challenges. In general, the analysis of the system model should determine whether the “knobs” have enough power (command authority) to actually drive the system in the required direction. Many other research questions remain open in the identification and modeling of the core phenomena to control in software systems. For example, *how to model explicitly and accurately the relationship among system goals, adaptation mechanisms, and the effects produced by controlled variables* when the control variables are as complex as a pattern and workflow? Can we design software systems having an explicit specification of what we want to assure with control-based approaches? Can we do it by focusing only on some aspects for which feedback control is more effective? Can we improve the use of control, or achieve control-based design, by connecting as directly as possible some real physics inside the software systems? How far can we go by modeling SAS systems mathematically? What are the limitations?

7.1.2 Sampling Period

One important decision when building a model is the sampling rate, that is, the frequency with which the states, outputs and commands are monitored and

processed. In terms of our models (1) and (2), what is the difference between time k and $k + 1$? The meaning of “1” above was “the next sample”, but after how many microseconds or minutes is the next sample? For models, such as (1) and (2), and for quantitative software qualities that can be approximated with continuous signals (performance, cost, energy), Nyquist-Shannon sampling theorem [50] can provide some practical guidelines. Simplifying, the theorem says that if the highest signal harmonic one wants to reconstruct from the sampling data has h Hertz, one should sample with a minimum frequency of $2h$ Hertz.

In Zheng *et al.* [66], the authors showed that for the specific performance model they were building, a sampling rate of five minutes was appropriate. Note that in [66] the authors were interested in mean response time and server provisioning decisions, which are rare control inputs. For higher frequency models and decisions (like changing the number of threads), that sampling rate is not enough. Further, Solomon *et al.* [53] showed how to find the sampling rate from the cutoff frequency. The cutoff frequency is the frequency of the command u that has no effect on the system (one can imagine that, for the example described by Eqs. (5) and (6), by adding and removing an HTTP connection at very high frequency, on average, has limited effect on the response time because the connection cannot be used). Solomon *et al.* use Bode plots to analyze models like (3) and (4) in the frequency domain and then determine the cutoff frequency. The sampling rate is then twice the cutoff frequency. It is obvious that the sampling rate depends on what we monitor and control, but it is unclear how to determine it in many cases and how to guarantee its correctness. Moreover, in SAS, most events are discrete and, thus, it makes more sense to use the number of events (such as number of failures or number of requests) instead of a sampling rate.

Assurance Challenges. The identification of the optimal sampling rate is a non-trivial task that can negatively impact the quality of the decisions taken by the controller. How can we ensure an optimal sampling rate over time? What is the overhead introduced by oversampling the underlying system? Can we control the sampling rate depending on the current state of the SAS?

7.2 Composition and Incrementality: V&V Tasks

Considering the scenarios we introduced for applying control theory to the engineering of SAS, validation and verification (V&V) is highly relevant, especially for the scenarios in which obtaining mathematical models is infeasible. If it is impossible to guarantee desirable properties based on the control theory principles and techniques, at least we should consider introducing V&V tasks on critical properties—an aspect that is commonly omitted in self-adaptive software proposals [58]. Nonetheless, performing V&V tasks (*e.g.*, model checking) over the entire system—at runtime, to guarantee desired properties and goals, is often infeasible due to prohibitive computational costs. Therefore, one fundamental requirement for the assurance of SAS systems is for these V&V tasks to be composable and applicable incrementally along the adaptation loop, among other conditions, as analyzed in [56].

Assurance Challenges. In this regard, relevant research questions include: *Which V&V tasks can guarantee which control properties*, if any, and to what extent? *Are stability, accuracy, settling-time, overshoot and other properties composable* (e.g., when combining control strategies which independently guarantee them)? What are suitable techniques to realize the composition of V&V tasks? Which approaches can we borrow from testing? How can we reuse or adjust them for the assurance of SAS systems? Regarding incrementality: in which cases is it useful? How can incrementality be realized? How do we characterize increments, and their relationship to system changes?

7.3 Timing Issues and Lags

As discussed previously, a major challenge in designing a control algorithm is to determine what is the largest control correction that does not destabilize the system. In addition to stability, SAS is affected by specific characteristics of the system dynamics, such as lags (when the system responds only slowly to a change in a tuning parameter), delays (when it takes time before a change takes any effect, such as startup time when adding a server), and differences between transient response (when response to a change in environment decays over time) and steady-state response (when the change is enduring). If the open and closed SAS are described by equations such as (3), the time lag should be included in the equations. The main problem in SAS is that lags are highly variable; it is hard, if not impossible, to bound them.

Another difference is that, in contrast to physical plants, behavior load in software systems sometimes presents no inertial effect. For example, in the room temperature control case, the trend of the difference between the reference level y and the measured output y_r usually augments or diminishes quite slowly. However, in software systems, all of the service requests can disappear instantly, for instance if all users cancel their requests, or the Internet connection of the web server is interrupted. Besides, while most physical processes do not respond immediately to changes in the control parameters, most software systems do not respond so precisely to control changes.

Even in the absence of a mathematical model, it is clear that the designer needs a good understanding of at least the direction of change that will result from a given control input, and preferably a sense of its magnitude. Since SAS systems often have more than one control input, the absence of a mathematical model makes understanding the relations among the inputs, disturbances, and lags more difficult, but no less important.

Lags, such as those that arise from bringing another server online, introduce some degree of uncertainty in SAS systems. Under such conditions, how can we assure the appropriate synchronization of control actions and SAS reactions? Can we guarantee the timing required by the software system to operate the change ordered by the control algorithm?

7.4 Challenges in Control Strategies Design

It is clear from the above discussion that decisions about the control strategy are absolutely critical to the design of SAS. Classical control theory operates in settings that are simple enough that mathematical analysis is often possible. SAS, on the other hand, are usually too complex to characterize completely, are highly non-linear and time-variant. This makes the design of the control strategy even more critical.

We often describe control in terms of strategic changes to the reference value or the level of external disturbances. It is almost impossible in SAS to design a controller that can work well with all possible values of references or disturbances. This is another reason to make explicit design decisions about the control strategy. When the system is time variant (that is, the matrices A, B, C, D, F in Eqs. (3) and (4) are time dependent) or the reference and disturbances change over large ranges, the design goals are achieved by engineering an adaptive controller that is tuned on-line (adaptive control), based on the operational point of the system. Adaptive control can be achieved using various strategies, such as Model Identification Adaptive Control (MIAC) or Model Reference Adaptive Control (MRAC) [38]. Those strategies are of paramount importance to SAS systems design.

7.4.1 Modeling External Disturbances and Uncertainties

In classical control theory, the feedback loop is designed with knowledge of the nature of the external disturbances, their characteristics and bounds. Furthermore, the design of the feedback system relies on sensed values about internal state or system behavior. This design must consider explicitly how accurately those sensed values actually represent the true state of the system. In the cruise control example, the control engineer knows that the speed of the car (the controlled output) is altered by friction, road conditions, hills and valleys, and all those can be captured in a model (like the term $F * p(k)$ in Eqs. (3) and (4)). Once a set point is set (*e.g.*, 100 km/h) the controller has to respond adequately (*i.e.*, with certain accuracy and overshoot) to changes in perturbations. These changes refer to both amplitude (the slope of the hill) and frequency (alternations of hills and valleys). Similar considerations are taken into account for thermostat control or autopilots.

In SAS, external disturbances are harder to identify and model. In addition, in SAS external disturbances are most likely to change, not the reference values, so we have to design feedback loops that respond well to external disturbances. Consider an application deployed in a public cloud or an application using services provided by third parties. Assuming a feedback loop that maintains the response time at a setpoint, what are the external disturbances that affect the response time? We can make some assumptions about application load (number of users or their distributions and requests during a day), but what about the external disturbances affecting the cloud or third party services? Indirectly those disturbances are going to affect the application response time. How do we

identify all those disturbances, their amplitude and frequency, how do we model or take their influence into account when we assure our feedback loop?

7.4.2 Complex Reference Values

It is tempting to set a reference level as a combination of goals. For example, to keep wait time below some threshold and energy consumption below some other threshold. Selecting a composite reference point exposes the risk of creating a situation in which one term of the reference value calls for increasing the tuning parameter and another term calls for decreasing the reference parameter. For example, if both wait time and energy consumption are limited, the former might lead to activating a server while the latter demands deactivating a server. In the case of the home thermostat, maintaining a comfortable temperature and saving energy consumption are clearly two goals that may lead to a conflictive situation. We must at minimum be aware of these conflicts; even better, we should refrain from creating them. In any event, thinking carefully about such conflicts can help avoid or manage the resulting complexity. One way to solve this is to use just one variable as a set point and use an optimization function like LQR to mediate among the other variables.

In the presence of complex reference values, can we detect such conflicting goals *a priori* or *a posteriori*? In case of conflicting goals, can we identify the constraints linking several goals in order to capture a more complex composite goal?

7.4.3 Management of Viability Zones

A viability zone of a SAS system can be defined as the set of possible states in which the system operation is not compromised with respect to a desired property [6]. Therefore, a SAS system may have more than one viability zone: at least one for the managed system and another for the controller or adaptation mechanism. The level of assurance of a SAS system may then be judged by taking into account the state of the system with respect to its viability zone(s). A viability zone can be characterized in terms of the properties to be satisfied, the quality attributes that describe them and corresponding desired values [59]. Moreover, viability zones can change according to variations in relevant context. Feedback control may also help in the specification of viability zones for SAS systems. Properties to be satisfied define the dimensions of a viability zone. These properties are usually characterized in terms of quality attributes that correspond to reference inputs (y_r) defining the boundaries of viability zones. The goal of adaptation mechanisms (commands u) is to maintain the managed system within its viability zone(s). Viability zones may be also dynamic, not only because relevant context and desired properties may change over time due to uncertainty, but also when the adaptation goal is optimization.

The definition of viability zones is not a trivial task, particularly because desired properties are usually characterized in terms of several variables (*i.e.*, quality attributes). Relevant research questions in this regard include: how many viability zones are required for the assurance of a particular SAS system? Does

each desired property require an independent viability zone? How to manage trade-offs and possible conflicts among several viability zones?

The dynamic nature of viability zones is also a relevant research challenge in the assurance of SAS systems. Particularly, because it implies adjusting the domain coverage not only of design and adaptation realization but also of V&V tasks. Open research questions in this aspect include: what are runtime models that can be used for the incremental and dynamic derivation of software artifacts for implementing V&V tasks? How to maintain the causal connection between viability zones, adapted system, and its corresponding V&V software artifacts? How to adapt these artifacts at runtime?

8 Conclusions

In this chapter, we explored control theory applications to self-adaptive software (SAS) design. We started with an overview of the SAS and control theory concepts, summarized the main research results of control theory applied to software and computing in general and then focused on control assurances. We described the properties of open loop models, the control strategies and goals, and then elaborated on assurance challenges.

There are several conclusions we can draw from this exploration. Control theory can be applied as is to a subset of SAS and for a subset of quality attributes, including performance, cost, reliability, energy consumption. Research results suggest that it is feasible to design a controller for relatively simple use cases, especially for single input/single output systems and for time invariant cases. Even for these metrics the research is still open, and more complex case studies are needed to draw general conclusions. To apply control theory, a specific type of model is needed. If a linear or discrete model can capture the outputs, control inputs, states, and eventually external perturbations of the open loop SAS, a controller can be synthesized.

Nonetheless, many complex SAS are time-variant and multi-dimensional, with multiple input and output control variables. For these cases, more advanced control theory concepts, such as Model Identification Adaptive Control (MIAC), are needed, where the model is identified at runtime and a controller is synthesized periodically. In even more cases, an explicit model for SAS cannot be constructed. Nevertheless, control theory general principles are still a good guideline to follow. The designer needs to understand the properties of open systems: inputs, outputs, perturbations, states, and how these influence each other. Properties, such as stability and controllability, should be investigated and understood before starting to design a controller (or adaptive/autonomic manager). The controller should be designed with some goals in mind: optimization, stability, performance and accuracy, and then tested and verified. A final conclusion is that the field of control theory applied to SAS and the related assurance challenges are still open, and constitute important research areas in the engineering of these kinds of software systems.

References

1. Abdelzaher, T., Stankovic, J., Lu, C., Zhang, R., Lu, Y.: Feedback performance control in software services. *IEEE Control Syst.* **23**(3), 74–90 (2003)
2. AlBahnassi, W., Mudur, S.P., Goswami, D.: A design pattern for parallel programming of games. In: 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), pp. 1007–1014. IEEE (2012)
3. Arcelli, D., Cortellessa, V., Filieri, A., Leva, A.: Control theory for model-based performance-driven software adaptation. In: Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, pp. 11–20. ACM (2015)
4. Åström, K.J., Murray, R.M.: *Feedback Systems. An Introduction for Scientists and Engineers* (2008)
5. Åström, K., Wittenmark, B.: *Adaptive Control*. Addison-Wesley series in Electrical Engineering: Control Engineering. Addison-Wesley, Reading (1995)
6. Aubin, J.P., Bayen, A.M., Saint-Pierre, P.: *Viability Theory: New Directions*. Springer, Heidelberg (2011). <https://doi.org/10.1007/978-3-642-16684-6>
7. Babaoglu, O., Canright, G., Deutsch, A., Caro, G.A.D., Ducatelle, F., Gambardella, L.M., Ganguly, N., Jelasity, M., Montemanni, R., Montresor, A., et al.: Design patterns from biology for distributed computing. *ACM Trans. Auton. Adapt. Syst. (TAAS)* **1**(1), 26–66 (2006)
8. Balzer, B., Litoiu, M., Müller, H., Smith, D., Storey, M.A., Tilley, S., Wong, K.: 4th International Workshop on Adoption-Centric Software Engineering. In: Proceedings of the 26th International Conference on Software Engineering, ICSE 2004, pp. 748–749. IEEE Computer Society, Washington, DC (2004)
9. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Saar, Y.: Synthesis of reactive (1) designs. *J. Comput. Syst. Sci.* **78**(3), 911–938 (2012)
10. Braberman, V., D’Ippolito, N., Kramer, J., Sykes, D., Uchitel, S.: MORPH: a reference architecture for configuration and behaviour self-adaptation. In: Proceedings of the 1st International Workshop on Control Theory for Software Engineering, CTSE 2015, pp. 9–16. ACM, New York (2015)
11. Bronstein, M.: Integration of elementary functions. *J. Symbolic Comput.* **9**(2), 117–173 (1990)
12. Brun, Y., et al.: Engineering self-adaptive systems through feedback loops. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) *Software Engineering for Self-Adaptive Systems*. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_3
13. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: *Pattern-oriented Software Architecture: A System of Patterns*. Wiley, New York (1996)
14. Cassandras, C., Lafontaine, S.: *Introduction to Discrete Event Systems*. Springer, New York (2008). <https://doi.org/10.1007/978-0-387-68612-7>
15. Checiu, L., Solomon, B., Ionescu, D., Litoiu, M., Iszlai, G.: Observability and controllability of autonomic computing systems for composed web services. In: Proceedings of the 2011 6th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI), pp. 269–274. IEEE (2011)
16. Cheng, B.H.C., et al.: Software engineering for self-adaptive systems: a research roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) *Software Engineering for Self-Adaptive Systems*. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_1

17. Clarke, E.M., Grumberg, O., Peled, D.: *Model Checking*. MIT Press, Cambridge (1999)
18. Dahm, W.: Technology Horizons a Vision for Air Force Science & Technology During 2010–2030. Tech. rep., U.S. Air Force (2010), <http://www.af.mil/information/technologyhorizons.asp>
19. Delaval, G., Gueye, S.M.K., Rutten, E., De Palma, N.: Modular coordination of multiple autonomic managers. In: Proceedings of the 17th International ACM SIGsoft Symposium on Component-Based Software Engineering, CBSE 2014, pp. 3–12. ACM, New York (2014)
20. Diao, Y., Gandhi, N., Hellerstein, J.L., Parekh, S., Tilbury, D.M.: Using MIMO feedback control to enforce policies for interrelated metrics with application to the apache web server. In: Network Operations and Management Symposium, NOMS 2002. IEEE/IFIP, pp. 219–234. IEEE (2002)
21. D’Ippolito, N., Braberman, V., Kramer, J., Magee, J., Sykes, D., Uchitel, S.: Hope for the best, prepare for the worst: multi-tier control for adaptive systems. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp. 688–699. ACM, New York (2014)
22. D’Ippolito, N., Braberman, V., Piterman, N., Uchitel, S.: Synthesizing nonanomalous event-based controllers for liveness goals. ACM Trans. Software Eng. Methodol. **22**(1), 9:1–9:36 (2013)
23. Elkhodary, A., Esfahani, N., Malek, S.: FUSION: a framework for engineering self-tuning self-adaptive software systems. In: Proceedings of 18th ACM International Symposium on Foundations of Software Engineering, FSE 2010, pp. 7–16. ACM (2010)
24. Filieri, A., Hoffmann, H., Maggio, M.: Automated design of self-adaptive software with control-theoretical formal guarantees. In: Proceedings of the 36th International Conference on Software Engineering, pp. 299–310. ACM (2014)
25. Filieri, A., Maggio, M., Angelopoulos, K., D’Ippolito, N., Gerostathopoulos, I., Hempel, A.B., Hoffmann, H., Jamshidi, P., Kalyvianaki, E., Klein, C., Krikava, F., Misailovic, S., Papadopoulos, A.V., Ray, S., Sharifloo, A.M., Shevtsov, S., Ujma, M., Vogel, T.: Software engineering meets control theory. In: Proceedings of 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015), pp. 71–82. IEEE Press (2015)
26. Gergin, I., Simmons, B., Litoiu, M.: A decentralized autonomic architecture for performance control in the cloud. In: 2014 IEEE International Conference on Cloud Engineering (IC2E), pp. 574–579. IEEE (2014)
27. Ghanbari, H., Litoiu, M., Pawluk, P., Barna, C.: Replica placement in cloud through simple stochastic model predictive control. In: Proceedings of the 2014 IEEE 7th International Conference on Cloud Computing (CLOUD), pp. 80–87. IEEE (2014)
28. Ghanbari, H., Simmons, B., Litoiu, M., Iszlai, G.: Feedback-based optimization of a private cloud. Future Gener. Comput. Syst. **28**(1), 104–111 (2012)
29. Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D.M.: *Feedback Control of Computing Systems*. Wiley, Chichester (2004)
30. IBM Corporation: An Architectural Blueprint for Autonomic Computing, Technical report, IBM Corporation (2006)
31. Janert, P.K.: *Feedback Control for Computer Systems*. O’Reilly Media Inc., Sebastopol (2013)

32. Kalyvianaki, E., Charalambous, T., Hand, S.: Self-adaptive and self-configured CPU resource provisioning for virtualized servers using kalman filters. In: Proceedings of the 6th International Conference on Autonomic Computing, pp. 117–126. ACM (2009)
33. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. *IEEE Comput.* **36**(1), 41–50 (2003)
34. Kramer, J., Magee, J.: Self-Managed systems: an architectural challenge. In: FOSE 2007: 2007 Future of Software Engineering, pp. 259–268. IEEE Computer Society, Washington, DC (2007)
35. Kusic, D., Kephart, J.O., Hanson, J.E., Kandasamy, N., Jiang, G.: Power and performance management of virtualized computing environments via lookahead control. *Cluster Comput.* **12**(1), 1–15 (2009)
36. Laddaga, R.: Guest Editor’s introduction: creating robust software through self-adaptation. *IEEE Intell. Syst.* **14**(3), 26–29 (1999)
37. Laddaga, R.: Active software. In: Robertson, P., Shrobe, H., Laddaga, R. (eds.) IWSAS 2000. LNCS, vol. 1936, pp. 11–26. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44584-6_2
38. de Lemos, R., et al.: Software engineering for self-adaptive systems: a second research roadmap. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 1–32. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_1
39. Lennart, L.: Perspectives on system identification. *Annu. Rev. Control* **34**(1), 1–12 (2010)
40. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming, 1st edn. Addison-Wesley Professional, Reading (2004)
41. Müller, H., Pezzè, M., Shaw, M.: Visibility of control in adaptive systems. In: Proceedings of the 2nd International Workshop on Ultra-Large-Scale Software-Intensive Systems, pp. 23–26. ACM (2008)
42. Müller, H., Villegas, N.: Runtime evolution of highly dynamic software. In: Mens, T., Serebrenik, A., Cleve, A. (eds.) Evolving Software Systems. LNCS, pp. 229–264. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-45398-4_8
43. Murray, R.M.: Control in an Information Rich World: Report of the Panel on Future Directions in Control, Dynamics, and Systems. SIAM, Philadelphia (2003)
44. Northrop, L., Feiler, P., Gabriel, R., Goodenough, J., Longstaff, T., Kazman, R., Klein, M., Schmidt, D., Sullivan, K., Wallnau, K.: Ultra-Large-Scale Systems—The Software Challenge of the Future. Technical report, Carnegie Mellon University Software Engineering Institute (2006), <http://www.sei.cmu.edu/uls>
45. Oreizy, P., Medvidovic, N., Taylor, R.N.: Runtime software adaptation: framework, approaches, and styles. In: Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), pp. 899–910 (2008)
46. Patikirikorala, T., Colman, A., Han, J., Wang, L.: A systematic survey on the design of self-adaptive software systems using control engineering approaches. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2012), pp. 33–42. IEEE Press (2012)
47. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.* **25**(1), 206–230 (1987)
48. Rutten, E., Marchand, N., Simon, D.: Feedback control as MAPE-K loop in autonomic computing. In: de Lemos, R., Garlan, D., Ghezzi, C., Giese, H. (eds.) Self-Adaptive Systems III. LNCS, vol. 9640, pp. 349–373. Springer, Cham (2016)
49. Salehie, M., Tahvildari, L.: Self-adaptive software: landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.* **4**, 14:1–14:42 (2009)

50. Shannon, C.: Communication in the presence of noise. Proc. IEEE **86**(2), 447–457 (1998)
51. Cowpertwait, Paul S.P., Metcalfe, Andrew V.: System Identification. In: Introductory Time Series with R. UR, pp. 201–209. Springer, New York (2009). https://doi.org/10.1007/978-0-387-88698-5_10
52. Solomon, B., Ionescu, D., Litoiu, M., Iszlai, G.: Autonomic computing control of composed web services. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010), pp. 94–103. ACM (2010)
53. Solomon, B., Ionescu, D., Litoiu, M., Iszlai, G., Prostean, O.: Measurements and identification of autonomic computing processes. In: Proceedings of the 2010 IEEE International Conference on Computational Intelligence for Measurement Systems and Applications (CIMSA), pp. 72–77. IEEE (2010)
54. Sykes, D., Corapi, D., Magee, J., Kramer, J., Russo, A., Inoue, K.: Learning revised models for planning in adaptive systems. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE 2013, pp. 63–71. IEEE Press, Piscataway (2013)
55. Tamura, G., Casallas, R., Cleve, A., Duchien, L.: QoS contract preservation through dynamic reconfiguration: a formal semantics approach. Sci. Comput. Program. (SCP) **94**(3), 307–332 (2014)
56. Tamura, G., et al.: Towards practical runtime verification and validation of self-adaptive software systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 108–132. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_5
57. Truex, D.P., Baskerville, R., Klein, H.: Growing systems in emergent organizations. Commun. ACM **42**(8), 117–123 (1999)
58. Villegas, N., Müller, H., Tamura, G., Duchien, L., Casallas, R.: A framework for evaluating quality-driven self-adaptive software systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011), pp. 80–89. ACM (2011)
59. Villegas, N.M.: Context Management and Self-Adaptivity for Situation-Aware Smart Software Systems. Ph.D. thesis, University of Victoria (2013)
60. Villegas, N.M., Tamura, G., Müller, H.A., Duchien, L., Casallas, R.: DYNAMICO: a reference model for governing control objectives and context relevance in self-adaptive software systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 265–293. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_11
61. Vromant, P., Weyns, D., Malek, S., Andersson, J.: On interacting control loops in self-adaptive systems. In: Proceedings of 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011), pp. 202–207. ACM (2011)
62. Wang, Y., Lafourche, S., Kelly, T., Kudlur, M., Mahlke, S.: The Theory of Deadlock Avoidance Via Discrete Control. Principles of Programming Languages. POPL, Savannah, USA, pp. 252–263 (2009)
63. Weyns, D., et al.: On patterns for decentralized control in self-adaptive systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 76–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_4

64. Zhang, Q., Zhu, Q., Zhani, M.F., Boutaba, R., Hellerstein, J.L.: Dynamic service placement in geographically distributed clouds. *IEEE J. Sel. Areas Commun.* **31**(12), 762–772 (2013)
65. Zheng, T., Woodside, M., Litoiu, M.: Performance model estimation and tracking using optimal filters. *IEEE Trans. Software Eng.* **34**(3), 391–406 (2008)
66. Zheng, T., Yang, J., Woodside, M., Litoiu, M., Iszlai, G.: Tracking time-varying parameters in software systems with extended Kalman filters. In: Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative research (CASCON), pp. 334–345. IBM Press (2005)

Evaluation

MCaaS: Model Checking in the Cloud for Assurances of Adaptive Systems

Amir Molzam Sharifloo and Andreas Metzger^(✉)

paluno (The Ruhr Institute for Software Technology),

University of Duisburg-Essen, Essen, Germany

{amir.molzamsharifloo, andreas.metzger}@paluno.uni-due.de

Abstract. Due to the uncertainty of what actual adaptations will be performed at run time, verifying adaptive systems at design time may lead to limited results or may even be infeasible. Run-time verification techniques have been proposed to cope with this uncertainty. Recently, there has been an increasing interest to use model checking (an important verification technique) at run time in order to verify the expected properties of adaptive systems. Given a system specification and expected system properties, a model checker determines whether or not the specification satisfies its properties in the presence of self-adaptation. One key concern is the generally high resource needs of model checking, which may prohibit its use on resource- and power-constrained devices, such as smart-phones or Internet-of-Things devices. To address this challenge, we introduce a cloud-based framework that delivers model checking as a service (MCaaS). MCaaS offloads computationally intensive model checking tasks to the cloud, thereby offering verification capabilities on demand. Adaptive systems running on any kind of connected device may take the advantages of model checking at run time by invoking the MCaaS service. To dynamically allocate the required cloud resources (CPU and memory), we employ machine learning to estimate the resource usage of an actual model checking task at run time. As proof of concept, we implement and validate the approach for the case of probabilistic model checking, which facilitates verifying typical properties such as reliability.

1 Introduction

Adaptive systems modify themselves at run time to react to context changes that – without adaptation – would lead to requirements violations. Examples of adaptive systems include mobile pervasive computing applications, Internet-of-Things devices, as well as cyber-physical systems, which have to respond to highly dynamic changes in their physical environments (e.g., see [20]). Responding to the need for addressing such dynamism, research activities on adaptive systems have intensified in the last decade (e.g., see [6, 7, 18, 22]).

An important concern is assurances of adaptive systems. In particular, assurances serve two main purposes:

- *Adaptation trigger*: Assurances are used to continuously check the satisfaction of requirements R against dynamic changes in the system’s context D' . Due to a dynamic change of a system’s context, the current system S may not be able to any longer satisfy its requirements. Assurances check whether the relationship $S, D' \models R$ hold and if not, trigger an adaptation of the system.
- *Correctness of adaptation*: Assurances are used to check that the planned adaptation steps will lead to a modified system S' that meets the requirements given the new context D' . This means assurances check whether the relationship $S', D' \models R$ holds.

One way to provide assurances at run time is to use run-time model checking (e.g., see [4, 5, 10, 23]). To do this, the behavior specification of the system is updated at run time to reflect context changes and system modifications. Model checking can verify if (1) the desired properties are violated due to context changes and thus whether there is a need for adaptation, and (2) it can check whether a modified specification will meet the properties.

Run-time model checking works well if sufficient computational resources (CPU, memory) are available. However, installing and running model checkers on resource- and power-constrained devices, such as smart-phones or Internet-of-Things devices is practically unfeasible. On the one hand, model checking tools are often only available for mainstream operating systems. For example, PRISM [17], the well-known probabilistic model checker, is available for Windows, Linux and OS X [1]. However, there is currently no support for Android and iOS, which are the main operating systems used by mobile devices. On the other hand, the limitations of processing power, memory and in particular energy (battery), severely limit the size and frequency of performing model checking tasks on the devices themselves.

Recent model checking advances have delivered light-weight techniques to reduce the run-time overhead by verifying the invariant parts of a specification at development time, when more resources and time are available (e.g., see [12, 15, 24]). The main assumption of these techniques is that the specification consists of invariant and variable parts, making it possible to isolate dynamic components, apply a kind of symbolic verification, and leave the rest of the analysis to run time where the missing information becomes available. This solution is reasonable and efficient for a class of applications in which the properties of interest and the system architecture are known at design time, whilst a sub-set of components may dynamically change at run time. However, for adaptive systems in which various components dynamically connect (such as cyber-physical systems), complete model checking is still required.

We propose exploiting cloud computing to make model checking available for resource- and power-constrained devices. Cloud computing offers elastic computational resources that are accessible remotely and on demand [11]. In particular, we introduce a cloud-based framework that delivers model checking *as a service* (MCaaS). MCaaS offloads computationally intensive model checking tasks to

the cloud, thereby offering verification capabilities on demand. The advantage of using cloud computing for delivering MCaaS lies in providing scalable computational power and memory based on the actual demand of a run-time model checking task. To dynamically allocate the required cloud resources (CPU and memory) for MCaaS, we employ machine learning to estimate the resource usage of an actual model checking task at run time. In addition, MCaaS facilitates the use of different kinds of model checkers through a single, standardized interface. For example, PRISM [17] supports the verification of Markov models against the probabilistic properties, while NuSMV [8] is a symbolic model checker that specializes the verification of classic temporal properties.

The remainder of this chapter is organized as follows. Section 2 introduces an example to motivate the application of model checking at run time. Section 3 elaborates our MCaaS framework, including the use of machine learning for estimating resource usage. Section 4 presents initial experimental results and discusses pros and cons of the MCaaS approach. Section 5 discusses related work. Section 6 concludes the chapter and provides and outlook to future work.

2 Motivating Example

To illustrate the relevance of run-time model checking, we chose a variation of the *TeleAssistance* case study presented in [2]. *TeleAssistance* is a medical assistance system designed for people affected by life-threatening diseases. The health state of patients is continuously monitored, and medical actions are provided accordingly. The system consists of sensors and a mobile device, carried by the patient. The mobile device in turn communicates with medical centers and hospitals. Figure 1(a) shows the workflow of the *TeleAssistance* system. Vital data (such as heart rate, blood pressure, and temperature) are continuously measured by sensors and sent to the mobile device, which then sends the data to one of the medical labs providing analysis services for the data. Depending on the analysis results, the *TeleAssistance* system may perform a drug change, set an alarm, or continue normally.

The *TeleAssistance* system uses 3rd party services, which are shown as activities with doubled-borders in Fig. 1. They are *Analyze Vital Data*, *Change Drug*, and *Set Alarm*. These services may be offered by different service providers, each of which offering different Quality of Service (QoS) levels. For instance, *Analyze Vital Data* may be provided by A-Lab with failure probability of 0.01 and cost of 50 cent/invocation, while B-Lab may offer this service with failure probability of 0.002 and cost of 75 cent/invocation. Moreover, the QoS levels of these services may change over time as, for instance, improved versions are offered by the service providers.

A typical requirement is that – due to the health condition of a certain patient – the *TeleAssistance* system must exhibit a reliability greater than 0.998. Run-time model checking allows us to check whether this requirements holds in the presence of an updated system specification or changes in QoS levels of 3rd party services. The updated specification is created at run time to respond to

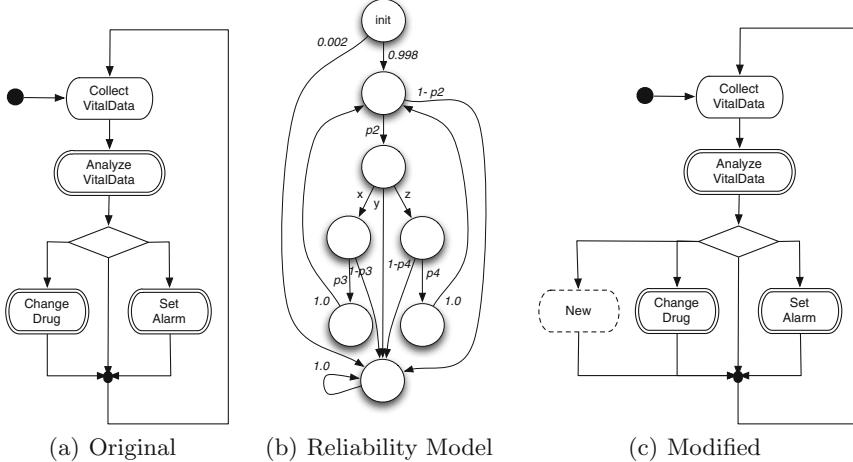


Fig. 1. Workflow of *TeleAssistance* system

changes that have been uncertain at design time. Furthermore, the workflow of the system may change at run time enabled by a flexible system design. Figure 1(c) shows an example, where an optional service may be added to the workflow to tackle a situation that was not identified earlier. Consequently, the corresponding reliability model needs to change and model checking needs to be carried out to check the updated version of the specification. Another case that may change the specification and thus triggers verification is when a new property is introduced on-the-fly. For example, a new reliability constraint or a property imposing the maximum cost of the workflow may be added by a user during run-time.

To check if the system satisfies the reliability requirement, a model checker may be employed as follows: A discrete-time Markov chain (DTMC) model is derived from the workflow of the system and is augmented with the probability values of the services. This leads to a reliability model as depicted in Fig. 1(b). The probability values are shown as parameters in the figure, but may actually be replaced with concrete QoS values for each service. By selecting one of the available service providers, the parameters appearing in the DTMC will be replaced with actual values and then we can check whether the model satisfies the expected reliability. To do this, the property – expressed in probabilistic computation tree logic (PCTL) – and the DTMC model are fed into a probabilistic model checker (such as PRISM) that can say whether or not the property holds.

3 MCaaS: Model Checking as a Service

Cloud computing provides compute resources (CPU, memory, network, software applications) as a service and makes them accessible through the Internet [11].

Elasticity is a key characteristic of cloud computing. Elasticity means that compute resources can be allocated, deployed, migrated and released dynamically according to demand. For instance, a cloud provider may instantiate or replicate a number of virtual machines on top of physical machines, and offer these virtual machines as infrastructure as a service. Thereby, cloud computing enables the construction of scalable systems that have to cope with varying and often unpredictable resource needs. Software developers enjoy the flexibility offered by clouds to have access to the necessary resources when they need them. A typical cloud computing use case is when the system has to respond to varying workloads. Without cloud computing, developers have to estimate in advance the maximum workload they may face and set up an appropriate computing environment that offers the resources required to meet the maximum workload. This has two main drawbacks: (1) the compute capacity allocated may be underutilized for most of the time, and (2) one is not able to react to peaks in workload that are above the estimated maximum workload. Cloud elasticity addresses these drawbacks, thereby reducing resource under-utilization as well as providing scalable resources in order to also meet peak demands.

In this section, we present the MCaaS framework, which provides a special form of Software-as-a-Service. MCaaS makes use of cloud elasticity to provide scalable model checking in the cloud.

3.1 MCaaS Overview

Figure 2 depicts the basic building blocks of the MCaaS framework. The framework follows a client-server architecture style. The provider (server) side is

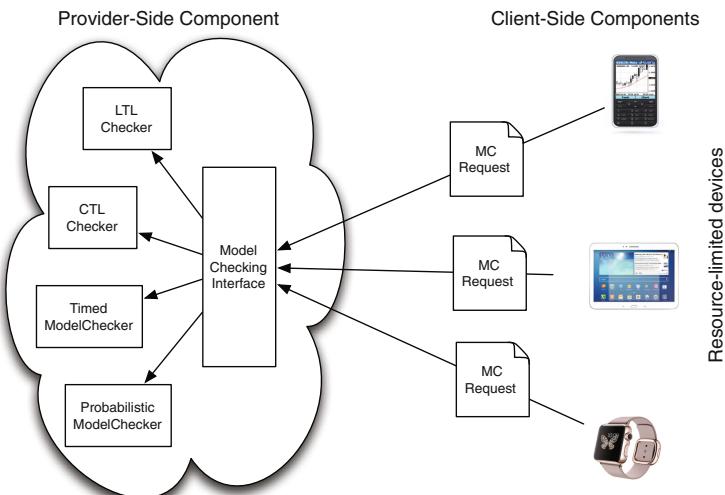


Fig. 2. Model checking as a service (MCaaS) framework

deployed in the cloud, whilst the client side resides on the resource-constrained devices.

To provide model checking capabilities best suited for the run-time model checking task, the server side offers different kinds of model checkers. To exploit MCaaS, the clients only need to invoke MCaaS by sending a model checking request (*MC Request*) to the MCaaS service endpoint. After successful completion of the model checking task, the client will receive the model checking results. In the following, we describe in more detail these client- and provider-side components.

3.2 MCaaS Client-Side Components

Let us consider the motivating example presented in Sect. 2. The mobile device running the *TeleAssistance* system continuously updates the reliability model with respect to the changes in external services and needs to assure that the updated model can satisfy the reliability requirement. Moreover, let us assume that two more requirements concerning cost and safety are specified and added at run time. These properties are expressed in Cost/Reward logic (see [17]) and CTL respectively. The satisfaction of these properties can be verified by a set of model checkers. One may choose PRISM to check probabilistic properties (PCTL and Cost/Rewards properties), and NuSMV for CTL, though these choices may vary depending on users' preferences.

Applying MCaaS, the client only needs to invoke MCaaS by sending a model checking request (*MC Request*). Each *MC Request* contains the following key pieces of information:

- the *kind of model checker* the client wants to run;
- the *model* to be checked, in a format compatible with what the chosen model checker expects;
- the *properties* to be checked;
- (optional) *constraints* for the model checking task, such as the precision of results, or the deadline within which the results should be returned.

Regarding our example, the reliability requirement is to be checked against the system model by using PRISM. In this case, we let PRISM set precision and do not consider any deadline. The request thus contains the following inputs:

- model checker = *PRISM*;
- model = *TeleAssistance.dtmc*;
- properties = {[F “final”] > 0.998}; The reliability requirement is expressed as a PCTL expression, which checks whether or not the probability of successful execution is greater than 0.998.
- constraints = \emptyset .

The concrete implementation of the *MC Request* is built upon the REST architectural style that simplifies the client-server interactions. More precisely, the communication is based on simple HTTP requests. HTTP is supported by

most modern clients and thus may be directly used to send the requests to the provider.

It should be noted that an *MC Request* does not include information about resource requirements. The resource requirements for each model checking task are estimated at runtime by the MCaaS provider-side components, as explained below.

3.3 MCaaS Provider-Side Components

The provider-side components include a front-end and the infrastructure back-end. The requests are received by the front-end which coordinates the virtual machines, on which the model checkers are deployed and which perform the actual computations. The front-end is the only element in the MCaaS framework that needs to keep running all the time.

As sketched in Fig. 3, the provider-side components follow a modular structure, which facilitates customization to the needs of service providers and clients. A minimal setup requires the presence of the module that dispatches the requests among the virtual machines (*Dispatch*). A more advanced front-end may include modules for client authentication and cryptography. Supported with these additional modules, the provider can guarantee the confidentiality of the computation requested by the clients. One may also envision to add modules to take care of subscriptions, billing and usage payments, which may be of interest in a commercial version of MCaaS. Further, as we introduce in Sect. 3.4, an *Online Learning* module is used to optimize resource estimation based on data obtained from past model checking tasks.

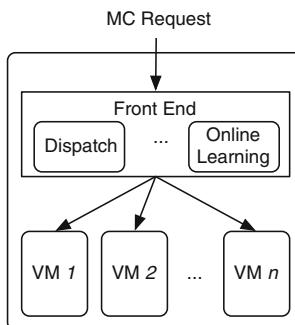


Fig. 3. MCaaS provider-side components

As soon as *MC Request* is received by the front-end, it is interpreted and sent to an appropriate virtual machine that provides the requested type of model checker. The amount of resources for the requested model checking task is estimated by the dispatcher (as explained in Sect. 3.4). In case a virtual machine

may not offer enough resources for the given model checking task, cloud elasticity is employed. This may lead to vertical or horizontal scaling. Vertical scaling means assigning more compute resources to the virtual machine. Horizontal scaling means adding virtual machines to distribute load – such as from different clients – across more virtual machines.

As vertical and horizontal scaling may take time (e.g., studies in [19] report of start-up times of up to 800 s), it may be more efficient and cost-effective to have a pool of virtual machines with different configurations ready to run rather than instantiating them only when needed. For these reasons, a practical implementation of MCaaS may have several virtual machines ready to execute model checking tasks, but may be able to scale when there is a high load on the system.

3.4 Resource Prediction

The resource demand for a model checking task highly varies depending on the specification structure, the properties to be checked, and the verification algorithm and its implementation. To provide an efficient MCaaS service, it is key to predict the required resources for a verification task as accurately as possible. To free the client from performing such prediction, MCaaS automatically predicts resource consumption on the provider side.

Our approach relies on machine learning to learn regression models, which express the relationship between different phenomena via mathematical functions. Given a data set of values for a set of variables V , regression produces a function \hat{f} to estimate a target variable $X = \hat{f}(V)$. Regression is a form of supervised learning, which means that the function \hat{f} is learned from a set of existing training data. In our case, such training data can either be historic or synthetic data available during design time (offline learning), or actual data collected during run time (online learning).

We perform *offline learning* before deploying the MCaaS provider-side components. For offline learning, a set of sample models is used to determine the training data set. Sample models are randomly generated to determine their attributes (V) and to experimentally measure the resource needs of the model checking task (X), in order to determine \hat{f} . The characteristics of the models depend on the domain and application area.

We perform *online learning* at run time to improve the accuracy of predictions. Online learning is employed to update the estimation function based on data collected by monitoring actual model checking tasks at run time.

During the execution of the MCaaS service, the learned functions (be it from offline or online learning) are used to predict the memory and execution time that a just-arrived request would demand. This information is used by *Dispatch* (see above) to distribute the model checking request to the matching virtual machine or to trigger resource scaling to meet the predicted resource demands.

Below we explain the steps involved for resource prediction in MCaaS. We start by describing the steps during design time and then describe the steps during run time.

Design-Time Steps of MCaaS: Figure 4 depicts the steps taken by MCaaS at design time in order to perform resource prediction at run time. It basically involves offline learning of prediction models.

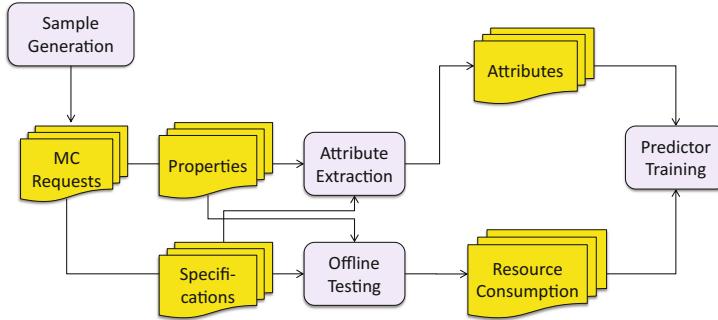


Fig. 4. Design-time steps of MCaaS (offline learning)

1. **Sample Generation.** Different sample requests are generated to be used for training the predictors. As is typical for machine learning, the more the training data represent the actual models that are requested for verification, the more precise the prediction would be [9]. Random models may be generated without any specific pattern, in case there is no knowledge about the characteristics of to-be-verified models. If the verification service is used for a specific domain, it is highly recommended to identify the common characteristics of potential models and narrow down the model generation to the models representing such characteristics. For example, as we will see in Sect. 4.2, for the reliability property of the TeleAssistance example, we generated random DTMC models with a certain number of states and transitions. As it may be challenging to generate a representative set of training data, we complement the offline learning part with online learning (see below).
2. **Attribute Extraction.** The first input of training a resource predictor is the characteristics of verification requests. Providing more information about the models and properties may increase the precision of run-time predictions. Feature extraction aims at identifying the important features (a.k.a attributes) in a request that includes the features of both the model and properties of interest. Feature extraction thus determines which data is most important for prediction. This can be done manually by a human or semi-automatically using techniques such as Principal Component Analysis [16]. For example, the number of states of a specification often has strong impact on memory consumption. The type of the property of interest is of importance as well, due to the fact that checking different class of properties has different costs. For example, verifying a temporal property that checks the next state is much cheaper than verifying a global safety property.

3. **Offline Testing.** To determine concrete resource needs for each sample model, the respective *MC requests* are executed on a machine instance with fixed (non-elastic) settings. The machine shall be computationally powerful to run the model checker required to verify a representative variety of *MC requests*. For each request, we generate a profile that contains the information on resource consumption such as memory and time. We facilitate testing by a monitoring tool that is able to continuously observe the resource consumption during the execution of the request. This information, together with the request attributes form the training data set.
4. **Predictor Training.** Having the request attributes and resource consumption profiles, we are ready to apply machine learning to produce the estimation functions. For each characteristic of interest (e.g., average memory consumption), an estimation function is learned as mentioned at the beginning of this section.

Run-Time Steps of MCaaS. During run time, upon the arrival of an *MC Request*, the front-end module decides which virtual machine is appropriate to accommodate it. To do so, the amount of resources demanded by the request is predicted. In addition, the estimation function is updated based on data collected by monitoring actual machine learning tasks (online learning). The respective steps are described below and shown in Fig. 5.

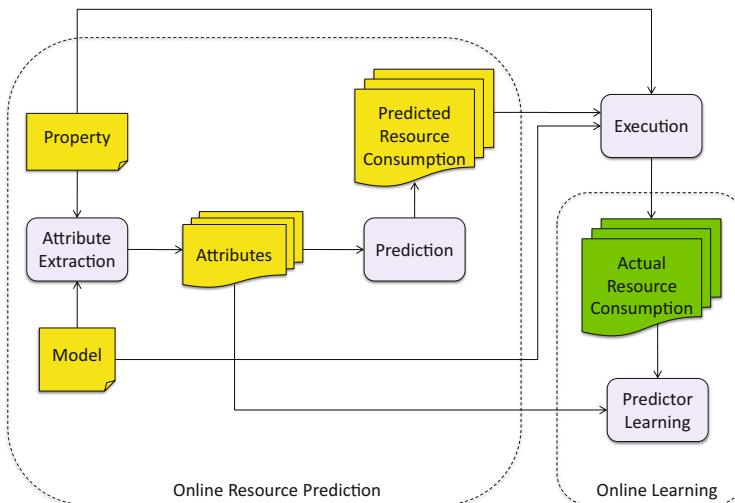


Fig. 5. Run-time steps of MCaaS

1. **Attribute Extraction.** Given an *MC request*, the values of important attributes are extracted from the property and system model provided as part of the *MC Request* message. These attributes have been determined during offline learning.

2. **Prediction.** The estimation functions are used to predict the resource consumption of a concrete *MC Request*. By feeding the values provided by attribute extraction to prediction functions, an approximate resource consumption profile is calculated.
3. **Execution.** According to the resources demanded for a request, an appropriate virtual machine instance is identified and tasked with the model checking request (or a new virtual machine is deployed if needed). The actual resource consumption of the concrete model checking task is monitored during its execution, and is used in the next step for improving the precision of future predictions.
4. **Predictor Learning.** The data collected by monitoring the resource consumption at run time may be used for further learning and for increasing prediction accuracy. The data may even be combined with data produced earlier during offline learning to produce a larger dataset. We recommend performing online learning periodically when there are substantial addition to the datasets. The output is then fed into Prediction for processing further requests.

4 MCaaS in Practice

This section presents our prototypical implementation of MCaaS, reports on initial experimental results obtained concerning resource consumption prediction, and critically discusses the overall MCaaS approach.

4.1 MCaaS Implementation

As proof of concept, an MCaaS prototype was implemented in Java. The MCaaS prototype provides a set of REST APIs that can be invoked by clients. Due to the large number of model checking techniques and tools, we focused on probabilistic model checking and in particular have chosen PRISM as a widely used tool to verify DTMC specifications and properties expressed in PCTL. However, the MCaaS framework and also its implementation is generic, facilitating incorporating support for other model checkers as well. PRISM is installed on different virtual machine instances with different configurations. Upon the arrival of a request, an appropriate virtual machine is selected and PRISM is invoked to serve the request. A monitoring process, also implemented in Java, is running in parallel to observe the resource consumption of the model checking task. As aforementioned, this information is recorded for further learning and prediction. For the machine learning part, we employed the Weka toolset¹.

To generate the training data set, we have implemented a Java program to randomly generate DTMCs. The program takes as input the characteristics of the desired models, e.g., number of states and transitions, and produces random DTMCs. The DTMCs are then checked by PRISM against a reliability property that is expressed as a reachability property in PCTL, while our monitoring tool captures the resource consumption, namely average memory and execution time.

¹ <http://www.cs.waikato.ac.nz/~ml/weka/>.

4.2 Accuracy of Resource Prediction

To demonstrate the applicability of MCaaS, we have experimentally assessed the capability of the machine learning component of MCaaS to accurately predict the resource needs on the provider side.

As training data, we generated 20 random DTMCs ranging between 1000 and 5000 states and having between 4 and 10 transitions per state. We ran PRISM for each of those models in order to measure the time and memory consumption. As representative verification task we chose a PCTL reachability property, which expresses the probability of reaching a failure state. The measurements were done on an isolated machine in order to control execution conditions and avoid uncontrolled effects. The specifications of the machine used for the experiment are as follows: OS = Mac, CPU = 2.4 GHz Core 2 Duo, and RAM = 4 GB. Note that for the sake of feasibility, we used rather small models and a low-end machine. As part of future work, we will perform larger-scale experiments to assess scalability of MCaaS and its accuracy in more challenging settings.

For the sake of simplicity, we considered only the number of states (S) and transitions (T) as important attributes. Based on S and T , the estimation functions for memory (Mem) and time ($Time$) that were learned from the samples are:

$$Mem = 0.0162 \times S + 5.2631 \times T + 26.6377$$

$$Time = 498.6432 \times S + 135619.5933 \times T - 1467870.1169$$

To assess accuracy of these predictors, we used another set of random DTMCs as test data. Figure 6 plots the predicted resource usage (memory and CPU time) against the actual resource usage. The results indicate that the estimation functions may predict resource consumption up to some degree of accuracy.

To quantify accuracy, we compute the mean absolute percentage error M (with \hat{f}_t as the predicted and f_t as actual value):

$$M = \frac{1}{n} \sum_{t=1}^n \left| \frac{f_t - \hat{f}_t}{f_t} \right|$$

The error for predicting memory consumption is $M_{mem} = 4.2\%$. The error for predicting CPU time is higher at $M_{CPU} = 45.3\%$.

To improve accuracy, a larger training data set may be required. Additionally, using more attributes during learning, such as considering the number of loops in the models, may increase accuracy. Finally, online learning, which exploits actual run-time data, may stabilize accuracy at a higher level.

4.3 Discussion

Model checking as a service (MCaaS), off-loads computationally expensive model checking tasks to the cloud and makes them accessible from resource-constrained devices through service interfaces. In this section we briefly discuss the advantages and limitations that come with MCaaS.

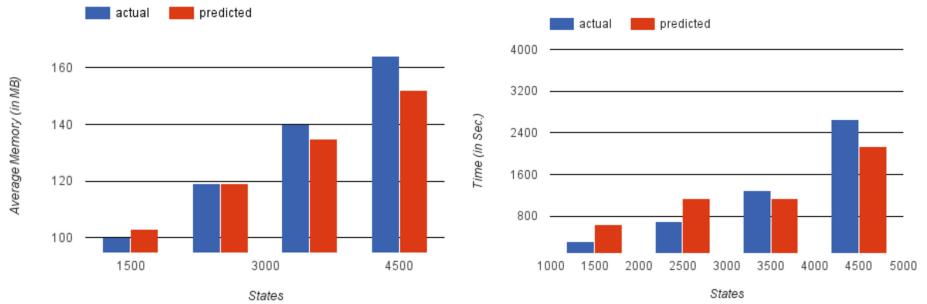


Fig. 6. Actual vs. predicted resource needs (memory and CPU time)

The main advantage of MCaaS is that it allows using model checking even from devices with limited memory and computational power. It offers the essential benefit of model checking techniques, which is the complete exploration of state space and the assurance that a specification meets a set of properties. This is different from other run-time verification approaches such as monitoring or online testing (e.g., see [21]) that rely on checking only some parts of the state space, without providing guarantees on property satisfaction. This specially matters in the case of dynamically adaptive systems where both specification and properties are subject to vary. By moving verification to run time, the system itself collects information and updates its specification and uses a run-time model checking service that can be used to analyze the specification and check the satisfaction of the properties, which may be derived from the requirements introduced lately at run time.

Of course, MCaaS on the provider side in principle faces the same limitations of model checking, such as state explosion and scalability to large models. MCaaS does not aim to improve this general concern of model checking. Yet, by resorting to dynamic resource provisioning in the cloud, MCaaS may provide better scalability than performing model checking on single (non-cloud) machines. However, even when offloading the model checking tasks to the cloud, the time required to model check a changed system specification before enacting the adaptation may become prohibitive. In the case of time-critical systems, their requirements towards real-time reaction may thus prohibit the use of MCaaS. In such case, other forms of assurances may be preferred, which however – as discussed above – do not offer the same advantages as model checking.

MCaaS works only when a device has access to the service through a network connection. We share the vision that in the future pervasive and mobile environments, most devices will be connected to different wireless or wired networks (e.g., see [20]). However, even then communication required between the MCaaS client and the provider may become a bottleneck, as communication may quickly drain power of resource-constrained devices. On the one hand, in the case of continuously adapting systems, the frequency and thus high number of model checking requests may become a concern and thus could limit the

applicability of MCaaS. On the other hand, the size of models can play a main role in consuming network bandwidth and power. Different solutions could be explored to avoid transferring a huge model each time a verification is needed. For example, an MCaaS provider may offer an additional storage service to keep the models of clients, while clients only need to send differences in models to the provider. As another example, incremental model checking approaches could be explored. Incremental approaches attempt to minimize the cost of model checking when run-time changes occur and thus may help reducing MCaaS requests and thereby communication overhead.

5 Related Work

In recent years, there has been an increasing interest to use model-checking techniques at run time. Epifani et al. [10] and Calinescu et al. [3,5] apply probabilistic model checking to evaluate and predict the QoS properties of service/component-based systems. They update the probabilistic specification of system with the online QoS monitored or predicted of each external service at run time, and then run PRISM [17] to check whether or not the expected QoS properties will be violated in the future.

Since running a model checker to verify the properties of a large specification can be very expensive, in terms of computation and memory usage, Filieri et al. [12] propose shifting the verification cost from run time to design time by applying parametric model checking. The main idea is to distinguish between QoS information of services, that may change, and those, which are invariant during system operation. By replacing the changing QoS information with variables and applying parametric model checking at run time, one is able to calculate a rational formula that can be evaluated at run time by feeding the real monitored/forecasted QoS to quickly and cheaply check the expected properties. However, the cost of parametric model checking grows exponentially as the number of parameters grows. In addition, the rational formula calculated by the model-checker may become very complex as the model scales up, which makes the run-time evaluations still expensive on resource-limited devices. Whilst Filieri et al.'s approach is very useful for specifications with invariant parts, with MCaaS we are capable of addressing more dynamic specifications, which in fact may completely change at run time. Moreover, MCaaS uses the existing model checkers as black box components and thus can immediately leverage enhancements of these model checking tools offered by the model checking community.

Concerning classic (non-quantitative) temporal properties, our previous work [24] presents a light-weight approach to check computation tree logic (CTL) properties of dynamic systems in which certain components remain unspecified at design time. A model-checking technique reasons about such incomplete specifications against a CTL property. The result of model checking includes a set of local properties only for the unspecified components such that if they are fulfilled by run-time components the satisfaction of the global property is guaranteed. At run time, the analysis is only applied to the unspecified components as soon

as they become available. The cost will be less than when model checking of the whole specification. However, even applying model checking to a sub-set of system components may not be reasonable for a resource-limited device.

Both of the above approaches [12] and [13, 14, 24] assume that the structure of the specification is invariant and only a set of components and services may change at run time. This assumption holds for a class of systems in which the system architecture is known at development time. However, for adaptive systems in which various components dynamically connect (such as cyber-physical systems), the system architecture may significantly vary and may only fully emerge at run time. Moreover, both approaches are applied to the properties that are introduced at development time. Following these approaches, the introduction of a new property that to be checked requires a further iteration of development and deployment activities. A more flexible and thus dynamic solution, which we enable with MCaaS tackles these issues by executing model checkers at run time remotely on the cloud.

6 Conclusion and Outlook

We presented the model checking as a service (MCaaS) framework that decouples applications requiring model checking tasks and the machines actually performing these tasks by leveraging cloud computing technology. Core element of the MCaaS framework is resource estimation supported by machine learning at both design time and run time. The MCaaS framework supports assurances of adaptive systems, by offering verification at run time to check the properties of interest, thereby addressing the uncertainty of how a system may actually adapt at run time.

Obviously, moving computation to the cloud does not eliminate the inherent complexity of a model checking problem, and so the exponential explosion of the state space still remains as an issue. However, MCaaS provides a means such that even resource-limited devices can take the advantages of these verification techniques.

As future work, we aim at expanding our experiments by a variety of model checking requests that require different model checking tools, ranging from classic to stochastic. We also plan to explore using a wider range of attributes during machine learning (such as structural complexity metrics) and investigate how these influence the prediction accuracy of the resource estimation for model checking tasks. In addition, we plan to investigate into a more flexible version of MCaaS, in which the kind of model checker does not have to be defined by the client but is automatically selected on the provider-side based on model characteristics and requested properties to be verified.

Acknowledgments. We cordially acknowledge the constructive comments provided by the anonymous reviewers. The research leading to these results has received funding from the European Community’s Seventh Framework Programme (FP7/2007–2013) under grant agreement 610802 (CloudWave).

References

1. Prism official website. <http://www.prismmodelchecker.org/>
2. Calinescu, R., Ghezzi, C., Kwiatkowska, M.Z., Mirandola, R.: Self-adaptive software needs quantitative verification at runtime. *Commun. ACM* **55**(9), 69–77 (2012)
3. Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G.: Dynamic QoS management and optimization in service-based systems. *IEEE Trans. Software Eng.* **37**(3), 387–409 (2011)
4. Calinescu, R., Kikuchi, S.: Formal methods @ runtime. In: Calinescu, R., Jackson, E. (eds.) *Monterey Workshop 2010*. LNCS, vol. 6662, pp. 122–135. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21292-5_7
5. Calinescu, R., Kwiatkowska, M.Z.: Using quantitative analysis to implement autonomic IT systems. In: *Proceedings of 31st International Conference on Software Engineering, ICSE 2009*, 16–24 May 2009, Vancouver, Canada, pp. 100–110 (2009)
6. Camara, J., de Lemos, R., Ghezzi, C., Lopes, A. (eds.): *Assurances for Self-Adaptive Systems*. LNCS, vol. 7740. Springer, Heidelberg (2013). <https://doi.org/10.1007/978-3-642-36249-1>
7. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.): *Software Engineering for Self-Adaptive Systems*. LNCS, vol. 5525. Springer, Heidelberg (2009). <https://doi.org/10.1007/978-3-642-02161-9> [outcome of a Dagstuhl Seminar]
8. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) *CAV 2002*. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
9. Domingos, P.: A few useful things to know about machine learning. *Commun. ACM* **55**(10), 78–87 (2012)
10. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: *ICSE*, pp. 111–121 (2009)
11. Erbes, J., Nezhad, H.R.M., Graupner, S.: The future of enterprise IT in the Cloud. *IEEE Comput.* **45**(5), 66–72 (2012)
12. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: *ICSE*, pp. 341–350 (2011)
13. Ghezzi, C., Menghi, C., Sharifloo, A., Spoletini, P.: On requirements verification for model refinements. In: *2013 21st IEEE International Requirements Engineering Conference (RE)*, pp. 62–71 (2013)
14. Ghezzi, C., Menghi, C., Sharifloo, A.M., Spoletini, P.: On requirement verification for evolving statecharts specifications. *Requirements Eng.* **19**(3), 231–255 (2013)
15. Ghezzi, C., Sharifloo, A.M.: Dealing with non-functional requirements for adaptive systems via dynamic software product-lines. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) *Software Engineering for Self-Adaptive Systems II*. LNCS, vol. 7475, pp. 191–213. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_8
16. Jolliffe, I.: *Principal Component Analysis*. Wiley Online Library, Chichester (2005)
17. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: probabilistic model checking for performance and reliability analysis. *ACM Perform. Eval. Rev.* **36**(4), 40–45 (2009)
18. de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.): *Software Engineering for Self-Adaptive Systems II*. LNCS, vol. 7475. Springer, Heidelberg (2013). <https://doi.org/10.1007/978-3-642-35813-5>

19. Mao, M., Humphrey, M.: A performance study on the VM startup time in the cloud. In: Chang, R. (ed.) 2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, 24–29 June 2012, pp. 423–430. IEEE (2012)
20. Metzger, A. (ed.): Cyber Physical Systems: Opportunities and Challenges for Software, Services, Cloud and Data. NESSI White Paper, February 2015
21. Metzger, A., Sammodi, O., Pohl, K.: Accurate proactive adaptation of service-oriented systems. In: Cámaras, J., de Lemos, R., Ghezzi, C., Lopes, A. (eds.) Assurances for Self-Adaptive Systems. LNCS, vol. 7740, pp. 240–265. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36249-1_9
22. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. TAAS 4(2) (2009). <https://doi.org/10.1145/1516533.1516538>
23. Schmieders, E., Metzger, A.: Preventing performance violations of service compositions using assumption-based run-time verification. In: Abramowicz, W., Llorente, I.M., Surridge, M., Zisman, A., Vayssi  re, J. (eds.) ServiceWave 2011. LNCS, vol. 6994, pp. 194–205. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24755-2_19
24. Sharifloo, A.M., Spoletini, P.: LOVER: light-weight formal verification of adaptive systems at run time. In: P  s  reanu, C.S., Sala  n, G. (eds.) FACS 2012. LNCS, vol. 7684, pp. 170–187. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35861-6_11

Analyzing Self-Adaptation Via Model Checking of Stochastic Games

Javier Cámar¹(✉), David Garlan¹, Gabriel A. Moreno², and Bradley Schmerl¹

¹ ISR - Institute for Software Research, Carnegie Mellon University,
Pittsburgh, PA 15213, USA

{jcmoreno,garlan,schmerl}@cs.cmu.edu

² SEI - Software Engineering Institute, Carnegie Mellon University,
Pittsburgh, PA 15213, USA
gmoreno@sei.cmu.edu

Abstract. Design decisions made during early development stages of self-adaptive systems tend to have a significant impact upon system properties at run time (e.g., safety, QoS). However, understanding the implications of these decisions *a priori* is difficult due to the different types and degrees of uncertainty that affect such systems (e.g., simplifying assumptions, human-in-the-loop). To provide some assurances about self-adaptive system designs, evidence can be gathered from activities such as simulations and prototyping, but these demand a significant effort and do not provide a systematic way of dealing with uncertainty. In this chapter, we describe an approach based on model checking of stochastic multiplayer games (SMGs) that enables developers to approximate the behavioral envelope of a self-adaptive system by analyzing best- and worst-case scenarios of alternative designs for self-adaptation mechanisms. Compared to other sources of evidence, such as simulations or prototypes, our approach is purely declarative and hence has the potential of providing developers with a preliminary understanding of adaptation behavior with less effort, and without the need to have any specific adaptation algorithms or infrastructure in place. We illustrate our approach by showing how it can be used to mitigate different types of uncertainty in contexts such as self-protecting systems, proactive latency-aware adaptation, and human-in-the-loop adaptation.

Keywords: Self-adaptation · Stochastic multiplayer games
Probabilistic model checking · Design-time assurances

1 Introduction

Complex software-intensive systems are increasingly relied upon in our society to support tasks in different contexts that are typically characterized by a high degree of uncertainty. Self-adaptation [13, 29] is regarded as a promising way to engineer in an effective manner systems that are *resilient* to runtime changes in their execution environment (e.g., resource availability), goals, or even in the system itself (e.g., faults).

Self-adaptive approaches typically focus on enforcing safety and liveness properties [4, 27] during operation, and optimizing qualities such as performance, reliability, or cost [26, 44]. However, providing actual *assurances* about the satisfaction of properties in a self-adaptive system is not easy in general, since its run-time behavior is largely influenced by the unpredictable behavior of the execution environment under which it is placed [23]. Moreover, providing assurances during early stages of the development process is of crucial importance, since major design decisions at that stage can have a significant impact on the properties of the resulting system.

When developers face the construction of a self-adaptive system, there is a large number of design decisions that need to be made before construction. For example, considerations such as whether to use reactive or proactive adaptation, whether to employ a centralized or decentralized decision-making approach, and whether it is possible to execute adaptations concurrently or one-at-a-time must be taken into account. Each of these considerations has different trade-offs, and results in vastly different self-adaptive system designs.

In general, the answer to these questions will be to a great extent informed by prior experience with similar existing systems, or by activities involving prototyping or simulation. However, experience with similar existing systems is not always available, and simulation and prototyping of (potentially many) system design variants is not cost-effective and does not provide systematic support to analyze a system (or its specification) in the context of an unpredictable environment.

Ideally, developers should be able to use techniques that provide some feedback about the implications of early design choices in the development process, helping them to narrow down the solution space in a cost-effective manner.

In this chapter, we propose an approach to analyze self-adaptive systems while accounting explicitly for the uncertainty in their operating environment, given some assumptions about its behavior. The approach enables developers to approximate the behavioral envelope of a self-adaptive system by analyzing best- and worst-case scenarios of alternative designs for self-adaptation mechanisms. The formal underpinnings of our proposal are based on model checking of stochastic multiplayer games (SMGs), which is a technique particularly suited to analyzing the interplay of a self-adaptive system and its environment, since SMG models are expressive enough to capture: (i) the uncertainty and variability intrinsic to the execution environment of the system in the form of probabilistic and nondeterministic choices, and (ii) the competitive behavior between the system and the environment, which can be naturally modeled as players in a game whose behavior is independent (reflecting the fact that changes in the environment cannot be controlled by the system).

The main idea behind the approach is modeling the system and its environment as players in a SMG that can either cooperate to achieve a common goal (for best-case scenario analysis), or compete against each other (for worst-case scenario analysis). The approach is purely declarative, employing adaptation knowledge based on architectural system descriptions as the scaffolding on

which game model specifications are built. The analysis of such SMG specifications enables developers to obtain a preliminary understanding of adaptation behavior, and it can be used to complement other sources of evidence that typically require the availability of specific adaptation algorithms and infrastructure, such as prototypes or simulations.

In [8] we reported on a concrete application of this technique to quantify the benefits of employing information about the latency of tactics for decision-making in proactive adaptation, comparing it against approaches that make the simplifying assumption of tactic executions not being subject to latency. In this chapter, we generalize our approach and show its versatility by describing how it can be instantiated for new applications to deal with other sources of uncertainty [23] (due to *parameters over time* and *human-in-the-loop*).

In the remainder of this chapter, Sect. 2 introduces some background and related work on different theories and approaches to support the construction of self-adaptive systems. Section 3 provides a general description of our approach, and Sect. 4 illustrates its application in different contexts, including the analysis of self-protecting systems, proactive latency-aware adaptation, and human-in-the-loop adaptation. Finally, Sect. 5 draws some conclusions and indicates directions for future work.

2 Background and Related Work

During the last few years, the research community has made an important effort in supporting the construction of self-adaptive systems. In particular, approaches to identify the added value of alternative design decisions have explored different theories (e.g., probability [31], fuzzy sets and possibility [42, 43], or games [35]) to factor in and mitigate the various types of uncertainty that affect self-adaptive systems.

Moreover, recent advances in formal verification [11, 33] have also explored the combination of probability and game theory to analyze systems in which uncertainty and competitive behavior are first-order elements.

This section overviews how different theories have been employed to analyze self-adaptive systems under uncertainty. We categorize the different approaches according to the theories employed and the different sources of uncertainty that they target conforming to the classification provided by Esfahani and Malek in [23] (Table 1).

2.1 Fuzzy Sets and Possibility Theory

Fuzzy set theory extends classical set theory with the concept of *degree of membership* [42], making its use appropriate for domains in which information is imprecise or incomplete. Rather than assessing the membership of an element to a set in binary terms (an element belongs to a set or not), fuzzy set theory describes membership as a function in the real interval $[0,1]$, where values closer to 1 indicate higher likelihood of the element belonging to the set. Possibility

Table 1. Theories and approaches to mitigate uncertainty in self-adaptive systems

Theory	Approach	Source of uncertainty	Simplifying assumptions	Model drift	Noise	Parameters over time	Human in the loop	Objectives	Decentralization	Context	Cyber-physical systems
Fuzzy sets/possibility theory	RELAX [39]						✓				
	FLAGS [2]						✓				
	POISED [22]	✓			✓ ^a		✓				
Probability theory	Rainbow [26]	✓		✓							
	FUSION [18]	✓		✓							
	Calinescu and Kwiatkowska [6]				✓						
	KAMI [21]				✓						
	QoS MOS [5]				✓						
	MAUS [24]					✓					
Probability + Game theory	Li et al. [34]						✓				
	Emami-Taba et al. [19]					✓					
	Camarra et al. [8,9]		✓			✓	✓				

^aPOISED employs probability theory to mitigate uncertainty due to noise.

theory [43] is based on fuzzy sets, and in its basic interpretation, it assumes that given a finite set (e.g., describing possible future states of the world), a *possibility distribution* is as a mapping between its power set, and the real interval $[0, 1]$ (i.e., any subset of the sample space has a possibility assigned by the mapping).

In the context of self-adaptive systems, possibility theory has been mainly used in approaches that deal with the uncertainty of the objectives [2, 22, 39]. RELAX [39] is a formal specification language for requirements that employs possibility theory to account for the uncertainty in the objectives of the self-adaptive system. The language introduces a set of operators that allows the “relaxation” of requirements at run time, depending on the state of the environment. FLAGS [2] also employs possibility theory to mitigate the uncertainty derived by the environment and changes in requirements by embedding adaptability at requirements elicitation. In particular, the framework introduces the concept of *adaptive goals* and *counter measures* that have to be executed if goals are not satisfied as a result of predicted uncertainty. POISED [22] is a quantitative approach that employs possibility theory to assess the positive and negative consequences of uncertainty. The approach incorporates a framework that can be tailored to specify the relative importance of the different aspects of uncertainty, enabling developers to specify a range of decision-making approaches, e.g., favoring adaptations that provide better guarantees in worst-case scenarios against others that involve higher risk but better maximization of the expected outcome.

2.2 Probability Theory

Probability theory [31] is the branch of mathematics concerned with the study of random phenomena. Probability is the measure of the likeliness that an event will occur, and is quantified as a number in the real interval $[0, 1]$ (where 0 indicates impossibility and 1 certainty). Within probability theory, frequentist interpretations of random phenomena employ information relative to the frequencies of past *actual* outcomes to derive probabilities that represent the *likelihood* of possible outcomes for future events.

This interpretation of probability is widely employed to deal with different sources of uncertainty in self-adaptive systems (e.g., context, simplifying assumptions, model drift) [18, 26]. FUSION [18] is an approach to constructing self-adaptive systems that uses machine learning to tune the adaptive behavior of a system in the presence of unanticipated changes in the environment. The learning focuses on the relation between adaptations, effects and system qualities, helping to mitigate uncertainty by considering explicitly interactions between adaptations. Rainbow [26] is an approach to engineering self-adaptive systems that includes constructs to deal with the mitigation of uncertainty in different activities of the MAPE loop [30]. In particular, the framework employs running averages to mitigate uncertainty due to noise in monitoring, as well as explicit annotation of adaptation strategies with probabilities (obtained from past observations of the running system) to account for uncertainty when selecting strategies during the planning stage.

Moreover, other approaches employ probabilistic verification and estimates of the future environment and system behavior for optimizing the system's operation. These proposals target the mitigation of uncertainty due to parameters over time [5,6,20]. Calinescu and Kwiatkowska [6] introduce an autonomic architecture that uses Markov-chain quantitative analysis to dynamically adjust the parameters of an IT system according to its environment and goals. Epifani *et al.* introduce KAMI [20], a methodology and framework to keep models alive by feeding them with run-time data that updates their internal parameters. The framework focuses on reliability and performance, and uses discrete-time Markov chains and queuing networks as models to reason about the evolution of non-functional properties over time. Moreover, the QoS MOS framework [5] extends and combines [6] and [20] to support the development of adaptive service-based systems. In QoS MOS, QoS requirements are translated into probabilistic temporal logic formulae used for identifying and enforcing optimal system configurations.

Eskins and Sanders introduce in [24] a Multiple-Asymmetric-Utility System Model (MAUS) and the opportunity-willingness-capability (OWC) ontology for classifying cyber-human systems elements with respect to system tasks. This approach provides a structured and quantitative means of analyzing cyber security problems whose outcomes are influenced by human-system interactions, dealing with the uncertainty derived from the probabilistic nature of human behavior.

2.3 Probability and Game Theory

Game theory is the study of mathematical models of conflict and cooperation between intelligent rational decision-makers [35]. The theory studies situations where there are multiple decision makers or *players* who have a variety of alternatives or *strategies* to employ in order to achieve a particular outcome (e.g., in terms of loss or payoff). Game theory has been applied in a wide variety of fields, such as, Economics, Biology, and Computer Science to study systems that exhibit competitive behavior (e.g., zero-sum games in which the payoff of a player is balanced by the loss of the other players), as well as a range of scenarios that might include cooperation (e.g., when players in a coalition coordinate to choose joint strategies by consensus).

Li et al. [34] present a formalism for human-in-the-loop control systems aimed at synthesizing semi-autonomous controllers from high-level temporal specifications (LTL) that expect occasional human intervention for correct operation. The approach adopts a game-theoretic approach in which controller synthesis is performed based on a (non-stochastic) zero-sum game played between the system and the environment. Although this proposal deals to some extent with uncertainty due to parameters over time and human involvement, the behavior of all players in the game is specified in a fully nondeterministic fashion, and once nondeterminism is resolved by a strategy, the outcome of actions is deterministic.

Emami-Taba et al. [19] present a game-theoretic approach that models the interplay of a self-protecting system and an attacker as a two-player zero-sum Markov game. In this case, the approach does not perform any exhaustive state-space exploration and is evaluated via simulation, emphasizing the learning aspects of the interaction between system and attacker.

Stochastic Multiplayer Games (SMGs) [11] are models that fit naturally systems that exhibit both probabilistic and competitive behavior among different players who can either collaborate with each other, or compete to achieve their own goals. In prior work, we presented in [8] an analysis technique based on model checking of SMGs to quantify the effects of simplifying assumptions in proactive self-adaptation. Specifically, the paper shows how the technique enables the comparison of alternatives that consider tactic latency information for proactive adaptation with those that are latency-agnostic, making the simplifying assumption that tactic executions are not subject to latency (i.e., that the duration of the time interval between the instants in which a tactic is triggered and its effects occur is zero). In [9] we adapted this analysis technique to apply it in the context of human-in-the-loop adaptation, extending SMG models with elements that encode an extended version of Stitch adaptation models [15] with OWC constructs [24].

We present in the following section SMGs in more detail, since they are the formal foundation that we employ in the approach described in this chapter.

2.4 Probabilistic Model Checking of Stochastic Multiplayer Games

Automatic verification techniques for probabilistic systems have been successfully applied in a variety of application domains including security [17, 37] and communication protocols [28]. In particular, techniques such as probabilistic model checking provide a means to model and analyze systems that exhibit stochastic behavior, effectively enabling reasoning quantitatively about probability and reward-based properties (e.g., about the system’s use of resources, time, etc.).

Competitive behavior may also appear in systems when some component cannot be controlled, and could behave according to different or even conflicting goals with respect to other components in the system. Self-adaptive systems are a good example of systems in which the behavior of some components that are typically considered as part of the environment (non-controllable software, network, human actors) cannot be controlled by the system. In such situations, a natural fit is modeling a system as a game between different players, adopting a game-theoretic perspective.

Our approach to analyzing self-adaptation builds upon a recent technique for modeling and analyzing stochastic multi-player games (SMGs) extended with rewards [11]. In this approach, systems are modeled as turn-based SMGs, meaning that in each state of the model, only one player can choose between several actions, the outcome of which can be probabilistic. This approach is particularly promising, since SMGs are amenable to analysis using model checking tools, and

are expressive enough to capture: (i) the uncertainty and variability intrinsic to the execution environment of the system in the form of probabilistic and non-deterministic choices, and (ii) the competitive behavior between the system and the environment, which can be naturally modeled as players in a game whose behavior is independent (reflecting the fact that changes in the environment cannot be controlled by the system).

Definition 1 (SMG). A turn-based stochastic multi-player game augmented with rewards (SMG) is a tuple $\mathcal{G} = \langle \Pi, S, A, (S_i)_{i \in \Pi}, \Delta, AP, \chi, r \rangle$, where Π is a finite set of players; $S \neq \emptyset$ is a finite set of states; $A \neq \emptyset$ is a finite set of actions; $(S_i)_{i \in \Pi}$ is a partition of S ; $\Delta : S \times A \rightarrow \mathcal{D}(S)$ is a (partial) transition function; AP is a finite set of atomic propositions; $\chi : S \rightarrow 2^{AP}$ is a labeling function; and $r : S \rightarrow \mathbb{Q}_{\geq 0}$ is a reward structure mapping each state to a non-negative rational reward. $\mathcal{D}(X)$ denotes the set of discrete probability distributions over finite set X .

In each state $s \in S$ of the SMG, the set of available actions is denoted by $A(s) = \{a \in A \mid \Delta(s, a) \neq \perp\}$. We assume that $A(s) \neq \emptyset$ for all states s in the model. Moreover, the choice of which action to take in every state s is under the control of a single player $i \in \Pi$, for which $s \in S_i$. Once action $a \in A(s)$ is selected by a player, the successor state is chosen according to probability distribution $\Delta(s, a)$.

Definition 2 (Path). A path of SMG \mathcal{G} is an (in)finite sequence $\lambda = s_0 a_0 s_1 a_1 \dots$ s.t. $\forall j \in \mathbb{N} \bullet a_j \in A(s_j) \wedge \Delta(s_j, a_j)(s_{j+1}) > 0$. The set of all finite paths in \mathcal{G} is denoted as $\Omega_{\mathcal{G}}^+$.

Players in the game can follow strategies for choosing actions in the game, cooperating with each other in coalition to achieve a common goal, or competing to achieve their own (potentially conflicting) goals.

Definition 3 (Strategy). A strategy for player $i \in \Pi$ in \mathcal{G} is a function $\sigma_i : (SA)^* S_i \rightarrow \mathcal{D}(A)$ which, for each path $\lambda \cdot s \in \Omega_{\mathcal{G}}^+$ where $s \in S_i$, selects a probability distribution $\sigma_i(\lambda \cdot s)$ over $A(s)$.

In the context of our approach, we always refer to player strategies σ_i that are *memoryless* (i.e., $\sigma_i(\lambda \cdot s) = \sigma_i(\lambda' \cdot s)$ for all paths $\lambda \cdot s, \lambda' \cdot s \in \Omega_{\mathcal{G}}^+$), and *deterministic* (i.e., $\sigma_i(\lambda \cdot s)$ is a Dirac distribution for all $\lambda \cdot s \in \Omega_{\mathcal{G}}^+$). Memoryless, deterministic strategies resolve the choices in each state $s \in S_i$ for player $i \in \Pi$, selecting actions based solely on information about the current state in the game. These strategies are guaranteed to achieve optimal expected rewards for the kind of cumulative reward structures that we use in our models.¹

Reasoning about strategies is a fundamental aspect of model checking SMGs, which enables checking for the existence of a strategy that is able to optimize an objective expressed as a property in a logic called rPATL. Concretely, rPATL can be used for expressing quantitative properties of SMGs, and extends the

¹ See Appendix A.2 in [11] for details.

logic PATL [12] (a probabilistic version of ATL [1], a logic extensively used in multi-player games and multi-agent systems to reason about the ability of a set of players to collectively achieve a particular goal). Properties written in rPATL can state that a coalition of players has a strategy which can ensure that the probability of an event’s occurrence or an expected reward measure meets some threshold.

rPATL is a CTL-style branching-time temporal logic that incorporates the coalition operator $\langle\langle C \rangle\rangle$ of ATL, combining it with the probabilistic operator $P_{\bowtie q}$ and path formulae from PCTL [3]. Moreover, rPATL includes a generalization of the reward operator $R_{\bowtie\bowtie}^r$ from [25] to reason about goals related to rewards. An extended version of the rPATL reward operator $\langle\langle C \rangle\rangle R_{\max=?}^r[F^* \phi]^2$ enables the quantification of the maximum accrued reward r along paths that lead to states satisfying state formula ϕ that can be guaranteed by players in coalition C , independently of the strategies followed by the rest of players. An example of typical usage combining the coalition and reward maximization operators is $\langle\langle \text{sys} \rangle\rangle R_{\max=?}^{\text{utility}}[F^c \text{ end}]$, meaning “value of the maximum utility reward accumulated along paths leading to an end state that a player sys can guarantee, regardless of the strategies of other players”.

3 Analysis of Self-Adaptation Via Model Checking of Stochastic Multiplayer Games

This section describes our approach to analyzing self-adaptive systems via model checking of SMGs. The approach enables developers to approximate the behavioral envelope of a self-adaptive system operating in an arbitrary environment, on which some assumptions are made. Figure 1 illustrates the underlying idea behind the approach, which consists in modeling both the self-adaptive system and its environment as two players of a SMG. The system’s player objective is optimizing an objective function encoded in a rPATL specification (e.g., minimizing the probability of violating a safety property, or maximizing accrued utility - encoded as a reward structure on the game). In contrast, the environment can either be considered as adversarial to the system (enabling worst-case scenario analysis), or as a cooperative player that helps the system to optimize its objective function (enabling best-case scenario analysis).

Our approach consists of two phases: (i) model specification, consisting in building the game model that describes the possible interactions between the self-adaptive system and its environment, and (ii) strategy synthesis, in which a game strategy that optimizes the objective function of the system player is built, enabling developers to quantify the outcome of adaptation in boundary cases.

² The variants of $F^* \phi$ used for reward measurement in which the parameter $\star \in \{0, \infty, c\}$ indicate that, when ϕ is not reached, the reward is zero, infinite or equal to the cumulated reward along the whole path, respectively.

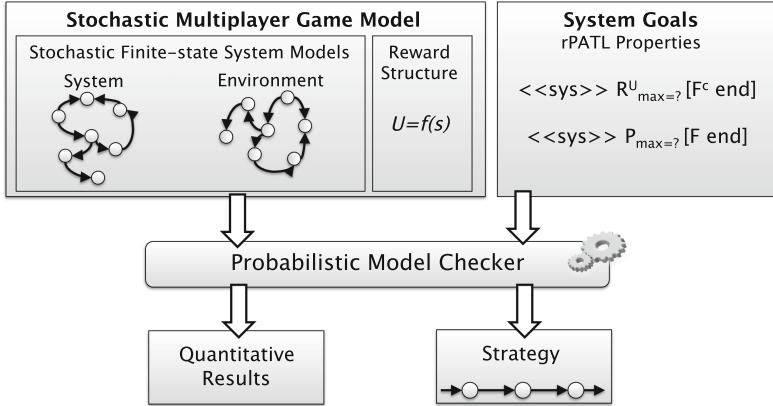


Fig. 1. Model checking of self-adaptation SMGs.

3.1 Model Specification

Model specification involves constructing the solution space for the system by specifying a stochastic multiplayer game $\mathcal{G} = \langle \Pi, S, A, (S_i)_{i \in \Pi}, \Delta, AP, \chi, r \rangle$, where:

- $\Pi = \{sys, env\}$ is the set of players formed by the self-adaptive system and its environment.
- $S = S_{sys} \cup S_{env}$ is the set of states, where S_{sys} and S_{env} are the states controlled by the system and the environment players, respectively ($S_{sys} \cap S_{env} = \emptyset$). States are tuples of values for state variables that capture system and environment state properties represented in architecture models. Moreover, a special state variable t encodes explicitly whether the current game state belongs to S_{sys} or S_{env} .
- $A = A_{sys} \cup A_{env}$ is the set of available actions in the game, where A_{sys} and A_{env} are the actions available to the system and the environment players, respectively.
- AP is a subset of all the predicates that can be built over the state variables.
- r is a reward structure labeling states with an associated cost or benefit. In the context of this chapter we assume a reward structure encoding utility. Specifically, the reward of an arbitrary game state s is defined as:

$$r(s) = \sum_{i=1}^q w_i \cdot u_i(v_i^s)$$

where u_i is the utility function for quality dimension $i \in \{1, \dots, q\}$, $w_i \in [0, 1]$ is the weight assigned to the utility of dimension i , and v_i^s denotes the value that quantifies quality attribute i in state s .

The state space and behaviors of the game are generated by performing the alphabetized parallel composition of a set of stochastic processes under the control of the system and environment players in the game (i.e., processes synchronize only on actions appearing in more than one process³):

System Player. The self-adaptive system (player *sys*) controls the choices made by two different processes:

- The *controller*, which corresponds to a specification of the behavior followed by the adaptation layer of the self-adaptive system, and can trigger the execution of tactics upon the target system under adaptation. The set of actions available to the controller process A_{sys} corresponds to the set of available tactics in the adaptation model. Each of these actions $a \in A_{sys}$ is encoded in a command of the form:⁴

$$[a] C_a \wedge \neg end \wedge t = sys \rightarrow t' = env$$

Where the guard includes: (i) the conjunction of architectural constraints that limit the applicability of tactic a (abstracted by C_a , e.g., a new server cannot be activated if all of them are already active), (ii) a predicate $\neg end$ to avoid expanding the state space beyond the stop condition for the game (abstracted by *end*), and (iii) a predicate to constrain the execution of actions of player *sys* to states $s \in S_{sys}$ (control of player turns is made explicit by a variable t that can take a value associated with any of the two players).

Note that in the most general case, all the local choices regarding the execution of controller actions are specified nondeterministically in the process, since this will enable the unfolding of all the potential adaptation executions when performing the parallel composition of the different processes in the model. However, the behavior of the controller can be further constrained to represent more specific instances of adaptation logic (e.g., as expressed in Stitch strategies) by including additional elements in the specification of the controller process. We illustrate both cases in the applications described in Sect. 4.

³ Please refer to <http://www.prismmodelchecker.org/doc/semantics.pdf> for further details.

⁴ We illustrate our approach to modeling the SMG using the syntax of the PRISM language [32], in which a process is encoded as a set of commands of the form:

$$[action] guard \rightarrow p_1 : u_1 + \cdots + p_n : u_n$$

Where *guard* is a predicate over variables in the model. Each update u_i describes a transition that the process can make (by executing *action*) if the guard is true. An update is specified by giving the new values of the variables, and has an assigned probability $p_i \in [0, 1]$. Multiple commands with overlapping guards (and commonly, including a single update of unspecified probability) introduce local nondeterminism.

- The *target system*, whose set of available actions is also A_{sys} . Action executions in the target system synchronize with those in the controller process on the same action names. In this case, each action can be encoded in one or more commands of the form:

$$[a] \text{pre}_a \rightarrow p_a^1 : \text{post}_a^1 + \cdots + p_a^n : \text{post}_a^n$$

$$[a] \text{pre}'_a \rightarrow p_a'^1 : \text{post}'_a^1 + \cdots + p_a'^n : \text{post}'_a^n$$

...

Hence, a specific action in the controller can synchronize non-deterministically with any of the alternative executions of the same action a in the target system. This models the different execution instances that the same tactic can have on the target system (e.g., when the controller enlists a server, the system can activate any of the alternative available servers). Each one of these commands is guarded by the precondition of a tactic’s execution instance, denoted by pre_a (e.g., a specific server needs to be disabled to be enlisted). Moreover, the execution of a command can result in a number of alternative updates on state variables (along with their assigned probabilities p_a^i) that correspond to the different probabilistic outcomes of a given tactic execution instance, denoted by post_a^i (e.g., the activation of a specific server can result in a successful boot with probability p , and fail with probability $1 - p$). Although strictly speaking, these probabilities describe part of the environment’s behavior by modeling probabilistic choices which are out of the system’s control, we chose to model them here as part of the target system to simplify presentation. In any case, note that since these choices are fully probabilistic, modeling them as part of the processes controlled by the environment player would not alter the outcome of the game.

Environment Player. The environment (player env) controls one or more stochastic processes that model potential disturbances in the execution context out of the system’s control such as network latency, or workload fluctuations. Each environment process is specified as a set of commands with asynchronous actions $a \in A_{env}$, and, similarly to the controller process, its local choices are specified nondeterministically to allow a broad range of environment behaviors within its possibilities. Each one of the commands follows the pattern:

$$[a] C_a^e \wedge \neg \text{end} \wedge t = env \rightarrow p_a^1 : \text{post}_a^1 \wedge t' = sys + \cdots + p_a^n : \text{post}_a^n \wedge t' = sys$$

Where C_a^e abstracts the set of environment constraints for the execution of action a (e.g., a threshold for the maximum latency that can be introduced in the network), and $\neg \text{end}$ prevents the generation of further states for the game. The command includes one or more updates, along with their associated probabilities. Each alternative update corresponds to one probabilistic outcome of the execution of a (post_a^i), and yields the turn to the system player.

3.2 Strategy Synthesis

Strategy synthesis consists of generating a memoryless, deterministic strategy in \mathcal{G} for player sys that optimizes its objective function. The specification of the objective for the synthesis of such strategy is given as a rPATL property that uses the operators $P_{\bowtie q}$ or $R_{\bowtie x}^r$ to optimize probabilities or rewards, respectively.

- *Probability-based properties* are useful to maximize the probability of satisfying a given property (or conversely, minimize the probability of property violation). We consider properties encoded following the pattern:

$$\langle\langle sys \rangle\rangle P_{\bowtie \in \{max=? , min=?\}} [F \phi]$$

An example of such property would be $\langle\langle sys \rangle\rangle P_{max=?} [F \text{ success}]$, meaning “value of the maximum probability to reach a `success` state that the system player can guarantee, regardless of the strategy followed by the environment”.

- *Reward-based properties* can help to maximize the benefit obtained from operating the system (e.g., in terms of utility), or minimize some cost (e.g., minimize the time to achieve a particular outcome when adapting the system).

We consider properties encoded using the pattern:

$$\langle\langle sys \rangle\rangle R_{\bowtie \in \{max=? , min=?\}}^r [F^* \phi]$$

In the context of our game specifications, the above pattern enables the quantification of the maximum/minimum accrued reward r along paths leading to states satisfying ϕ that can be guaranteed by the system player, independently of the strategy followed by the environment. Examples of such properties are $\langle\langle sys \rangle\rangle R_{max=?}^{\text{utility}} [F^c \text{ empty_batt}]$ (“value of the maximum accrued utility reward that the system can guarantee before the full depletion of the battery, regardless of the strategy followed by the environment”), or $\langle\langle sys \rangle\rangle R_{min=?}^{\text{time}} [F^c rt < \text{MAX_RT}]$ (“value of the minimum time that the system can guarantee to reach an acceptable performance level in which response time rt is below a threshold `MAX.RT`, regardless of the strategy followed by the environment”).

In the next section, we illustrate our approach by describing how the checking of rPATL properties on SMG models can support the analysis of self-adaptive behavior in different contexts.

4 Applications

In this section, we first illustrate how to instantiate our approach in the context of self-protecting systems, comparing how different variants of system defense mechanisms perform in an environment including attackers. Second, we describe how the approach can be employed to reason about the effects of latency in adaptation. Finally, we describe an application of SMG analysis to systematically reason about the involvement of humans in the execution of adaptations in the context of an industrial middleware used to monitor energy production plants.

Our formal models are implemented in PRISM-games [10], an extension of the probabilistic model-checker PRISM [32] for modeling and analyzing SMGs.

4.1 Self-protecting Systems

Probabilistic model checking of SMGs can be a powerful tool applied in the context of self-protecting systems [40]. In this context, the interplay between a defending system and a potentially hostile environment including attackers can be modeled as competing players in a SMG. In this section, we illustrate how model checking of SMGs can be used to model variants of self-protecting systems to compare their effectiveness. Concretely, we compare a generic version of Moving Target Defense (MTD) [36] with the use of self-adapting software architectures [41].

The fundamental premise behind MTD is to create a dynamic and shifting system, which is more difficult to attack than a static system. To be considered an MTD system, a system needs to shift the different vectors along which it could potentially be attacked. Such a collection of vectors is often termed an attack surface, and changing the surface in different ways as the system runs makes an attack more difficult because the surface is not fixed. The main motivation behind an MTD system is to significantly increase the cost to adversaries attacking it, while avoiding creating a higher cost to the defender.

In contrast, self-adapting software architectures tackle in a different way the self-protection of software systems by applying specific strategies to increase the complexity of the system, decrease the potential attack surface, or aid in the detection of attacks [38, 41]. In terms of MTD, self-adaptive systems apply approaches at the architectural or enterprise level, meaning that security can be reasoned about in the context of other qualities and broader business concerns.

The results at the end of this section demonstrate the impact of shifting self-protecting mechanisms from working in response to existing stimulus (reactive) to acting in preparation for potential perceived threats based on predictions about the environment (proactive). We consider in our study three variants of self-protection:

- *Uninformed-Proactive*. The defending system adapts proactively with a fixed time frequency based on an internal model of the environment (i.e., it does not factor in sensed information about the environment in decision making regarding when or which tactics should be performed).
- *Predictive-Proactive*. The system adapts proactively, but factoring in information sensed from the environment, as well as predictions about the environment’s future behavior (e.g., trend analysis, or seasonal information).
- *Reactive*. The defending system adapts reactively, executing tactics based on information sensed from the environment (e.g., after a number of detected probing events that can increase the amount of information that a potential attacker might have available, thereby increasing its chances of carrying out a successful attack).

Game Description. Our formal model for analyzing self-protecting systems is played in turns by two players that are in control of the behavior of the environment (including an attacker), and the defending system, respectively.

In this game, the environment's goal is compromising the defending system by carrying out an attack on it. The probability of success of the attack is directly proportional to the amount of information that the attacker has successfully gathered about the system through subsequent probing attempts during the game. On the other hand, the goal of the defending system is thwarting the attacks by adapting the system. The behavior of the system includes a single, abstract adaptation tactic that has the effect of invalidating the information that the attacker has collected about the system up to the point in which the system adapts. Probing, attacking, and adapting are actions that incur costs in terms of consumed resources, both on the attacker's and the defending system's sides.

- *System Player.* The behavior of the defending system is parameterized by the constants shown in Table 2 (note that `MAX_THREAT_LEVEL` and `THREAT_SENSITIVITY` are relevant only to the reactive game variant). During its turn, the system can:
 - Yield the turn to the environment player without executing any actions.
 - Adapt, resulting in the (partial) invalidation of the information collected by the attacker. The adaptation of the system can only be executed if the following conditions are satisfied:
 - * There must be enough available system resources to carry out the adaptation (`ADAPTATION_COST`).
 - * The perceived threat level must be above the threshold (`THREAT_SENSITIVITY`). This condition applies only in the reactive variant of the

Table 2. Constants parameterizing the behavior of the system player.

Name	Game variant	Description
<code>MAX_SYSTEM_RES</code>	All	Maximum amount of available system resources
<code>ADAPTATION_COST</code>	All	Amount of resources consumed each time the system adapts
<code>ADAPTATION_EFFECTIVENESS</code>	All	Probability of invalidating the information that the attacker has gathered about the system
<code>MAX_THREAT_LEVEL</code>	Reactive	Maximum level of threat as perceived by the defending system
<code>THREAT_SENSITIVITY</code>	Reactive	Degree of reactivity of the defending system regarding threat detection (e.g., it is the minimum threshold in perceived threat level required to adapt, where threat level is increased by external events such as attacks or probes)

model (the value of the threshold is always set to zero in proactive variants).

- * The system should be able to adapt at the current time. This applies only for the uninformed proactive variant of the model, in which the system is allowed to adapt only with a fixed frequency (`ADAPT_PERIOD`).

Once the adaptation commands executes, it carries out a reduction in the amount of information collected by the attacker directly proportional to the value set in the parameter `ADAPTATION_EFFECTIVENESS`. In the reactive version of the system, the level of perceived threat is also reduced in the same proportion.

- *Environment Player*. The behavior is parameterized by the constants shown in Table 3. During its turn, the environment player can either:

- Probe the system if there are enough resources available for it. There are two probabilistic outcomes for the probing action: (i) the probe succeeds with probability `P_PROBE_SUCCESS`, incrementing the amount of information available to the attacker, as well as the threat level perceived by the system, or (ii) the probe fails with probability `1-P_PROBE_SUCCESS`, incrementing only the threat level perceived by the system.
- Attack the system if there are enough resources available for it. The possible outcomes of the attack are: (i) the attack succeeds with probability `P_ATTACK_SUCCESS`, and (ii) the attack fails with probability `1-P_ATTACK_SUCCESS`, raising the value of the threat level perceived by the system.
- Yield the turn to the system player without executing any actions.

Table 3. Constants parameterizing the behavior of the environment player.

Name	Description
<code>MAX_ATTACKER_RES</code>	Maximum amount of resources available to the attacker
<code>PROBE_COST</code>	Amount of resources consumed when probing the system
<code>ATTACK_COST</code>	Amount of resources consumed when attacking the system
<code>PROBE_THREAT_DELTA</code>	Increment in perceived threat level caused by a probe on the system
<code>ATTACK_THREAT_DELTA</code>	Increment in perceived threat level caused by an attack on the system
<code>PROBE_INFO_GAIN</code>	Amount of information obtained from successfully probing the system
<code>P_PROBE_SUCCESS</code>	Probability that a probe on the system will successfully obtain useful information for the attacker

Analysis Results. To compare the different variants of self-protecting mechanisms, we carried out a set of experiments in which we checked the minimum probability of compromising the system that each of the defense variants could guarantee, independently of the strategy followed by the environment/attacker. This corresponds to quantifying the rPATL property, where `compromised` is a predicate that encodes the occurrence of a successful attack event in the game:

$$P_{Comp} \triangleq \langle\langle sys \rangle\rangle P_{min=?}[\mathbf{F} \text{ compromised}]$$

We instanced all the variants of the game model with the set of parameters values displayed in Table 4, exploring how P_{Comp} evolved throughout the range of values [5, 50] for available system resources, with the rest of the parameter values fixed.

Table 4. General parameter values for model instantiation.

System		Environment/Attacker	
MAX_SYSTEM_RES	[5, 50]	MAX_ATTACKER_RES	5
ADAPTATION_COST	1	PROBE_COST	0
ADAPTATION_EFFECTIVENESS	1	ATTACK_COST	1
MAX_THREAT_LEVEL	5	ATTACK_THREAT_DELTA	2
		PROBE_THREAT_DELTA	1
		PROBE_INFO_GAIN	1
		P_PROBE_SUCCESS	0.8

Specifically, we carried out two experiments:

- *Comparison of uninformed vs. predictive variants of proactive adaptation.* Figure 2 shows a comparison of the maximum probability that the attacker has of compromising the system for the two variants that implement proactive defense. The uninformed variant adapts with the maximum possible frequency allowed by the available amount of resources to the system (e.g., if the amount of available resources is 5, and the time frame defined for the game is 50, the system will adapt every 10 time units). The results show that given the same amount of system resources, the predictive variant always performs better than uninformed adaptation.⁵ Moreover, while the predictive variant progressively and smoothly reduces the probability of the attacker compromising the system, the uninformed one is more uneven, presenting different intervals during which the addition of system resources does not make any difference concerning the probability of the system being compromised (e.g., the probability does not change for the uninformed variant during the interval [25, 49]).

⁵ Our experiments assume that the predictive variant has access to a perfect prediction of the future evolution of the environment.

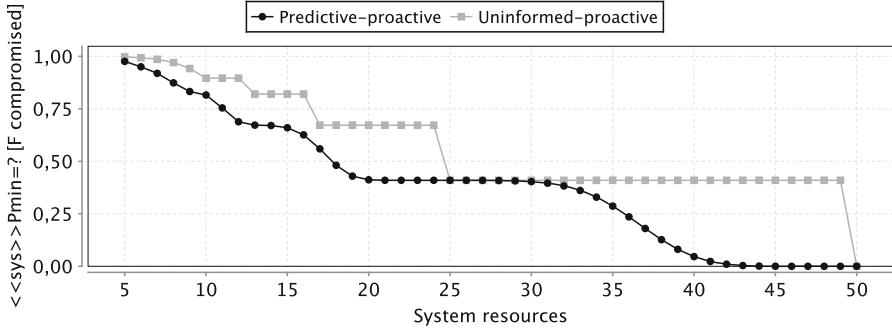


Fig. 2. Probability of compromising the system in proactive adaptation: uninformed *vs.* predictive.

- Comparison of predictive proactive adaptation *vs.* reactive adaptation. Figure 3 shows P_{Comp} for the predictive variant of MTD, comparing it with reactive adaptation variants that have different threat sensitivity levels. As expected, predictive proactive adaptation always performs better than the different reactive variants, since the solution space of reactive adaptation is always a subset of the solutions available to predictive proactive adaptation. In particular, it can be observed how increasing levels of sensitivity (i.e., lower values of threshold THREAT_SENSITIVITY) yield increasingly better results.

In this section, we have shown how SMG analysis can be used to compare alternative self-protection mechanisms in the presence of an adversarial environment. We quantify the performance of each alternative by analyzing their worst-case scenario against an attacker, even in the presence of some degree of uncertainty about the future behavior of the environment (i.e., due to *parameters over time*).

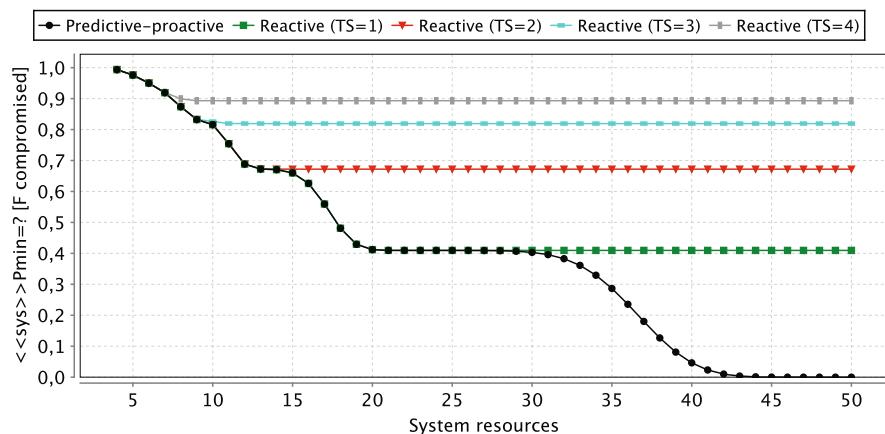


Fig. 3. Probability of compromising the system in proactive *vs.* reactive adaptation.

In the next section, we show how the approach is used to handle uncertainty due to *simplifying assumptions*.

4.2 Latency-Aware Proactive Adaptation

When planning how to adapt, self-adaptive approaches tend to make simplifying assumptions about the properties of adaptation, such as ignoring the time it takes for an adaptation tactic to cause its intended effect. Different adaptation tactics take different amounts of time until their effects are produced. For example, consider two tactics to deal with an increase in the load of a system: reducing the fidelity of the results (e.g., less resolution, fewer elements, etc.), and adding a computer to share the load. Adapting the system to produce results with less fidelity may be achieved quickly if it can be done by changing a simple setting in a component, whereas powering up an additional computer to share the load may take some time. We refer to the time it takes since a tactic is started until its effect is achieved as *tactic latency*. Current approaches to decide how to self-adapt do not take the latency of adaptation tactics into account when deciding what tactic to enact. For proactive adaptation, considering tactic latency is necessary so that the adaptation can be started with the sufficient lead time to be ready in time.

In this section, we show how SMG analysis can help to handle the uncertainty derived from *simplifying assumptions* by enabling the comparison of adaptation alternatives that consider tactic latency information for proactive adaptation with those that are latency-agnostic.

Game Description. We model our game for analysis of latency-aware adaptation based on [Znn.com](#) [14], a case study portraying a representative scenario for the application of self-adaptation in software systems which has been extensively used to assess different research advances in self-adaptive systems. [Znn.com](#) embodies the typical infrastructure for a news website, and has a three-tier architecture consisting of a set of servers that provide contents from backend databases to clients via front-end presentation logic (Fig. 4). The system uses a load balancer to balance requests across a pool of replicated servers, the size

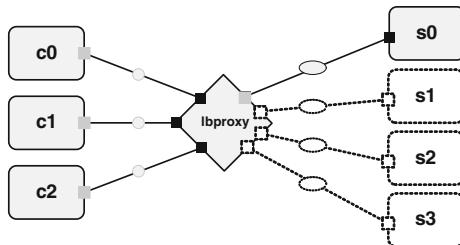


Fig. 4. [Znn.com](#) system architecture

of which can be adjusted according to service demand. A set of clients makes stateless requests, and the servers deliver the requested contents.

The main objective for [Znn.com](#) is to provide content to customers within a reasonable response time, while keeping the cost of the server pool within a certain operating budget. It is considered that from time to time, due to highly popular events, [Znn.com](#) experiences spikes in requests that it cannot serve adequately, even at maximum pool size. To prevent losing customers, the system can maintain functionality at a reduced level of fidelity by setting servers to return only textual content during such peak times, instead of not providing service to some of its customers. Concretely, there are two main quality objectives for the self-adaptation of the system: (i) performance, which depends on request response time, server load, and network bandwidth, and (ii) cost, which is associated with the number of active servers.

In [Znn.com](#), when response time becomes too high, the system is able to increment its server pool size if it is within budget to improve performance; or switch servers to textual mode if the cost is near to budget limit.⁶

The game is played in turns by two players that are in control of the behavior of the environment and the system, respectively. We assume that the frequency with which the environment updates the value of non-controllable variables, and the system responds to these changes is defined by the constant TAU.⁷ Hence, two consecutive turns of the same player are separated by a time period of duration TAU.

- *Environment Player.* The environment is in control of the evolution of time and other variables of the execution context that are out of the system’s control (e.g., service requests arriving at the system). During its turn, the environment sets the amount of request arrivals for the current time period and updates the values of other environment variables (e.g., increasing the variable that keeps track of execution time).
- *System Player.* During its turn, the system can trigger the activation of a new server, which will become effective only after the latency period of the tactic expires (modeled using a counter that keeps track of time since the tactic was triggered). Alternatively, the system can discharge a server (with no latency associated). Concretely, the system can execute one of the following actions during its turn:
 - Triggering the activation of a server. This action executes only if the current number of active servers has not reached the maximum allowed, and the counter that controls tactic latency is inactive (meaning that there is not currently a server already booting in the system). Triggering server activation sets the value of the counter to the latency value for the tactic.

⁶ We consider a simple version of [Znn.com](#) that adapts only by adjusting server pool size.

⁷ TAU is the period of the self-adaptation control loop that includes the monitoring of the environment, and the adaptation decision.

- Effective server activation, which executes only when the counter that controls tactic latency reaches zero, incrementing the number of servers in the system, and deactivating the counter.
- Deactivation of a server, which decrements the number of active servers. This action is executed only if the current number of active servers is greater than the minimum allowed and the counter for server activation is not active.
- Yield the turn to the environment player without executing any actions (decreasing the value of the latency counter if it is active).

In addition, the system updates during its turn the value of the response time according to the request arrivals placed by the environment player during the current time period and the number of active servers (computed by a M/M/c queuing model [16]).

The objective of the system player in the game is maximizing the accrued utility during execution. To represent utility, we employ a reward structure that maps game states to a single instantaneous utility value computed according to a set of utility functions and preferences (i.e., weights). Specifically, we consider two functions that map the response time and the number of active servers in the system to performance and cost utility, respectively.

In latency-aware adaptation, the reward structure rIU encoded in the game employs the real value of response time and number of servers during the tactic latency period to compute the value of instantaneous utility. However, in non-latency-aware adaptation, the instantaneous utility expected by the algorithm during the latency period for activating a server does not match the real utility extracted for the system, since the new server has not yet impacted the performance. In this case, we add to the model a new reward structure $rEIU$ in which the utility for performance during the latency period is based on the response time that the system would have if the new server had completed its activation.

Analysis Results. To compare latency-aware *vs.* non-latency-aware adaptation, we make use of rPATL specifications that enable us to analyze (i) the maximum accrued utility that adaptation can guarantee, independently of the behavior of the environment (worst-case scenario).

- *Latency-aware adaptation.* We define the *real guaranteed accrued utility* (U_{rga}) as the maximum real instantaneous utility reward accumulated throughout execution that the system player is able to guarantee, independently of the behavior of the environment player:

$$U_{rga} \triangleq \langle\langle \text{sys} \rangle\rangle R_{\max=?}^{rIU}[F^c t = \text{MAX_TIME}]$$

This enables us to obtain the utility that an optimal self-adaptation algorithm would be able to extract from the system, given the most adverse possible conditions of the environment.

- *Non-latency-aware Adaptation.* In non-latency-aware adaptation, real utility does not coincide with the expected utility that an arbitrary algorithm would employ for decision-making, so analysis is carried out in two steps:
 1. Compute the strategy that the adaptation algorithm would follow based on the information it employs about expected utility. That strategy is computed based on an rPATL specification that obtains the expected guaranteed accrued utility (U_{ega}) for the system player:

$$U_{ega} \triangleq \langle\langle \text{sys} \rangle\rangle R_{\max=?}^{\text{rEIU}} [F^c t = \text{MAX_TIME}]$$

For the specification of this property we employ the expected utility reward rEIU instead of the real utility reward rIU.⁸

2. Verify the U_{rga} under the generated strategy. We do this by building a product of the existing game model and the strategy synthesized in the previous step, obtaining a new game under which further properties can be verified. In our case, once we have computed a strategy for the system player to maximize expected utility, we quantify the reward for real utility in the new game in which the system player strategy has already been fixed.

Table 5 compares the results for the utility extracted from the system by a latency-aware *vs.* a non-latency-aware version of the system player, for two different models of [Znn.com](#) that represent an execution of the system during 100 and 200 s, respectively. The models consider a pool of up to 4 servers, out of which 2 are initially active. The period duration TAU is set to 10 s, and for each version of the model, we compute the results for three variants with different latencies for the activation of servers of up to 3*TAU s. The maximum number of arrivals that the environment can place per time period is 20, whereas the time it takes the system to service every request is 1 s.

The relative difference between the expected and the real guaranteed utility is defined as:

$$\Delta U_{er} = (1 - \frac{U_{ega}}{U_{rga}}) \times 100$$

Moreover, we define the relative difference in real guaranteed utility between latency-aware and non-latency aware adaptation as:

$$\Delta U_{rga} = (1 - \frac{U_{rga}^n}{U_{rga}^l}) \times 100,$$

where U_{rga}^n and U_{rga}^l designate the real guaranteed accrued utility for non-latency-aware and latency-aware adaptation, respectively.

The results show that latency-aware adaptation outperforms in all cases its non-latency-aware counterpart. Concretely, latency-aware adaptation is able to guarantee an increment in utility extracted from the system, independently of

⁸ Note that for latency-aware adaptation $U_{ega} = U_{rga}$.

Table 5. SMG model checking results for Znn

MAX_TIME (s)	Latency (s)	Latency-aware			Non-latency-aware			ΔU_{rga} (%)
		U_{ega}	U_{rga}	$\Delta U_{er}(\%)$	U_{ega}	U_{rga}	$\Delta U_{er}(\%)$	
100	TAU	53.77	53.77	0	65.97	48.12	-27.05	10.5
	2*TAU	49.35	49.35	0	64.3	42.1	-34.5	14.69
	3* TAU	45.6	45.6	0	64.3	33.25	-48.2	27
200	TAU	110.02	110.02	0	127.25	95.9	-24.63	12.83
	2*TAU	105.6	105.6	0	125.57	76.6	-38.99	27.46
	3* TAU	101.17	101.17	0	123.9	66.15	-46.6	34.61

the behavior of the environment (ΔU_{rga}) that ranges between approximately 10 and 34%, increasing progressively with higher tactic latencies. Regarding the delta between expected and real utility that adaptation can guarantee, we can observe that ΔU_{er} is always zero in the case of latency-aware adaptation, since expected and real utilities always have the same value, whereas in the case of non-latency-aware adaptation there is a remarkable decrement that ranges between 24 and 48%, also progressively increasing with higher tactic latency.

4.3 Human-in-the-Loop Adaptation

The different activities of the MAPE-K loop in some classes of systems (e.g., safety-critical) and application domains can benefit from human involvement by: (i) receiving information difficult to automatically monitor or analyze from humans acting as sophisticated sensors (e.g., indicating whether there is an ongoing anomalous situation), (ii) incorporating human input into the decision-making process to provide better insight about the best way of adapting the system, or (iii) employing humans as system-level effectors to execute adaptations (e.g., in cases in which full automation is not possible, or as a fallback mechanism).

However, the behavior of human participants is typically influenced by factors external to the system (e.g., training level, stress, fatigue) that determine their likelihood of succeeding at carrying out a particular task, how long it will take, or even if they are willing to perform it in the first place. Without consideration of these factors, it is difficult to decide when to involve humans in adaptation, and in which way.

Answering these questions demands new approaches to systematically reason about how the behavior of human participants, incorporated as integral system elements, affects the outcome of adaptation. In this section, we illustrate how the explicit modeling of human factors in SMGs can be used to analyze human-system-environment interactions to provide a better insight into the trade-offs of involving humans in adaptation, handling explicitly the uncertainty due to *human-in-the-loop*. In particular, we focus on the role of human participants as actors (i.e., effectors) during the execution stage of adaptation, and illustrate the

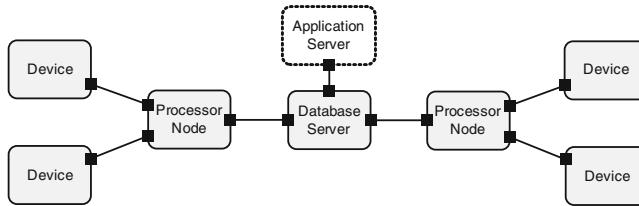


Fig. 5. Architecture of a DCAS-based system.

approach in the context of DCAS (Data Acquisition and Control Service) [7], a middleware from Critical Software that provides a reusable infrastructure to manage the monitoring of highly populated networks of devices equipped with sensors.

The basic building blocks in a DCAS-based system (Fig. 5) are:⁹

- *Devices* are equipped with one or more sensors to obtain data from the application domain (e.g., from wind towers, or solar panels). Each sensor has an associated *data stream* from which data can be read. Each type of device has its particular characteristics (e.g., data polling rate, or expected value ranges) specified in a *device profile*.
- *Processor nodes* pull data from the devices at a rate configured in the device profile, and dispatch this data to the database server. Each processor node includes a set of processes called *Data Requester Processor Pollers* (DRPPs or *pollers*, for short) responsible for retrieving data from the devices. Communication between DRPPs and devices is synchronous, so DRPPs remain blocked until devices respond to data requests or a timeout expires. This is the main performance bottleneck of DCAS.
- *Database server* stores the data collected from devices by processor nodes.
- *Application server* is connected to the database server to obtain data, which can be presented to the operators of the system or processed automatically by application software. However, DCAS is application-agnostic, so the application server will not be discussed in the remainder of this paper.

The main objective of DCAS is to collect data from the connected devices at a rate as close as possible to the one configured in their device profiles, while making an efficient use of the computational resources in the processor nodes. Specifically, the primary concern in DCAS is providing service while maintaining acceptable levels of performance, measured in terms of processed data requests per second (rps) inserted in the database, while the secondary concern is optimizing the cost of operating the system, which is mapped to the number of active processor nodes (i.e., each active processor node has a fixed operation cost per time unit).

⁹ We herein consider a simplified version of the DCAS architecture. Further details about DCAS can be found in [7].

In situations in which new devices are connected to the network at runtime and all available resources in the set of active processor nodes are already being used, DCAS includes a scale out mechanism aimed at maintaining an acceptable performance level by dynamically activating new processor nodes, according to the demand determined by the new system's workload and operating conditions. Scale out can be performed in two different ways:

- As a *manual process* carried out by a human operator. This is a slow and demanding process, in which a new processor node must be manually deployed, and devices re-attached across the different already active processor nodes, according to the particular situation.
- As an *automated process* that can be executed by adaptation logic residing in an closed control loop. Although this process is faster than the manual version of scale out and does not require any human intervention, it is less effective in terms of exploiting system resources, since only new devices can be attached to the new (pre-deployed) processor nodes being activated (i.e., devices already attached to other processor nodes cannot be re-attached, restricting the space of target configurations that the system can adopt with respect to the manual variant of scale out).

Game Description. Our SMG model for DCAS is aimed at enabling us to discriminate situations under which the involvement of human actors for the execution of adaptations is advisable. In particular, the game is tailored to devise the best possible outcome (in terms of accrued utility) that the system can guarantee in the worst-case scenario (both with and without human involvement). The game is played in turns by two players that are in control of the behavior of the environment and a DCAS-based system, respectively. Concretely, the system player controls of all the actions that belong to a human actor and the target system, including the execution of adaptation tactics for manual and automatic scale out. The objective of the system player is maximizing accrued utility obtained from performance and cost during execution.

- *Environment Player.* Controls the evolution of variables in the execution context that are out of the system's control. For the sake of simplicity, we assume in this case a neutral behavior of the environment that only keeps track of time, although additional behavior controlling other elements (e.g., network delay) can be encoded (please refer to Sects. 4.1 and 4.2 for further details illustrating the modeling of adversarial environment behavior).
- *System Player.* Models the cooperative behavior of the system and the human operator. It consists of two parts:
 - *Human model.* Attributes of human actors that might affect interactions with the system are captured in a model inspired by an opportunity-willingness-capability (OWC) ontology described in the context of cyber-human systems [24]. Although a single actor is modeled in our game, system descriptions can incorporate multiple human actor types (e.g., human actor roles specialized in different tasks), each of which can have

multiple instances (e.g., operators with different levels of training in a particular task). Attributes of human actor types can be categorized into:

- * *Opportunity.* Captures the applicability conditions of the adaptation tactics that can be executed by human actors upon the target system, as constraints imposed on the human actor (e.g., by the physical context – is there an operator physically located on site?).

Example 1. We consider a tactic to have a human actor manually deploy a processor node (`addPN`) when performing scale out in DCAS. Opportunity elements are $OE^{\text{addPN}} = \{L, B\}$, where L represents the operator's location, and B tells us whether the operator is busy doing something else:

- $L.\text{state} \in \{\text{operator on location (ONL)}, \text{operator off location (OFFL)}\}$.
- $B.\text{state} \in \{\text{operator busy (OB)}, \text{operator not busy (ONB)}\}$.

Using OE^{addPN} , we can define an opportunity function for the tactic $f_O^{\text{addPN}} = (L.\text{state} == \text{ONL}) \cdot (B.\text{state} == \text{ONB})$ that can be used to constrain its applicability only to situations in which there is an operator on location who is not busy.

- * *Willingness.* Captures transient factors that might affect the disposition of the operator to carry out a particular task (e.g., load, stamina, stress). Continuing with our example, willingness elements in the case of the `addPN` tactic can be defined as $WE^{\text{addPN}} = \{S\}$, where $S.\text{state} \in [0, 10]$ represents the operator's stress level. A willingness function mapping willingness elements to a probability of tactic completion can be defined as $f_W^{\text{addPN}} = pr_W(S.\text{state})$, with $pr_W : S \rightarrow [0, 1]$.
- * *Capability.* Captures the likelihood of successfully carrying out a particular task, which is determined by fixed attributes of the human actor, such as training level. In our example, we define capability elements as $CE^{\text{addPN}} = \{T\}$, where T represents the operator's level of training (e.g., $T.\text{state} \in [0, 1]$). We define a capability function that maps training level to a probability of successful tactic performance as $f_C^{\text{addPN}} = pr_C(T.\text{state})$, with $pr_C : T \rightarrow [0, 1]$.
- *Target System Behavior.* Models the execution of the different tactics on the system. During its turn, the system player can execute two actions per tactic available. We focus on tactic `addPN` to illustrate how tactic execution is modeled:
 - * Tactic trigger happens when: (i) an operator is on location and not busy, (ii) the number of active processor nodes is lower than the maximum available, and (iii) the latency counter for the tactic is zero. As a consequence, the operator is flagged as busy and the latency counter is activated.
 - * Tactic completion. When the tactic's latency counter expires, the command can either: (i) update variables for performance and active number of processor nodes according to a successful activation of a

processor node with probability `addPN.wc_prob` (determined by the willingness and capability elements defined in the human model), or (ii) fail to activate the node with probability $1 - \text{addPN.wc_prob}$. In both cases, the latency counter is reset, and the busy status of the operator is set to false.

In addition, the system player can choose not to execute any actions, just updating any tactic active latency counters that have not reached their respective tactic latency values. Note that the encoding used for the automatic scale out tactic (`activatePN`) follows the same structure, but without any OWC elements encoded in the guards or updates of the commands (activation of a processor node using this tactic is assumed to be always successful).

- *Adaptation Logic.* The adaptation logic placed in the controller is modeled as two alternative adaptation strategies¹⁰ for scale out (`scaleOutOp` and `scaleOut`). Actions in the specification of the strategies synchronize with the trigger actions on the specification of the target system behavior on shared action names.
 - * `scaleoutOp` models the variant of the scale out mechanism that makes use of a human actor by first triggering tactic `addPN`. Once an observation period for the effects of the tactic has expired, the strategy falls back on automatic activation by triggering the `activatePN` tactic if the activation of the new processor node by the human operator has not been successful.
 - * `scaleOut` models the automatic scale out mechanism with a single command that triggers the execution of tactic `activatePN` on the target system.

The game includes an encoding of utility functions and preferences for performance and cost as a reward structure `rGU` that enables the quantification of instantaneous utility in game states.

Analysis Results. We exploit our human-in-the-loop adaptation game model to deal with the uncertainty derived from involving humans in adaptation, employing its analysis to determine: (i) the expected outcome of human involvement in adaptation, and (ii) the conditions under which such involvement improves over fully automated adaptation.

- *Strategy Utility.* The expected utility value of an adaptation strategy (potentially including non-automated tactics) is quantified by checking the reachability reward property:

$$u_{mau} \triangleq \langle\langle \text{sys} \rangle\rangle R_{\max=?}^{\text{rGU}} [F^c t = \text{MAX_TIME}]$$

¹⁰ In this context we refer to adaptation strategies as described in Stitch [15]. These correspond to decision trees in which branches are defined by means of condition-action-delay rules.

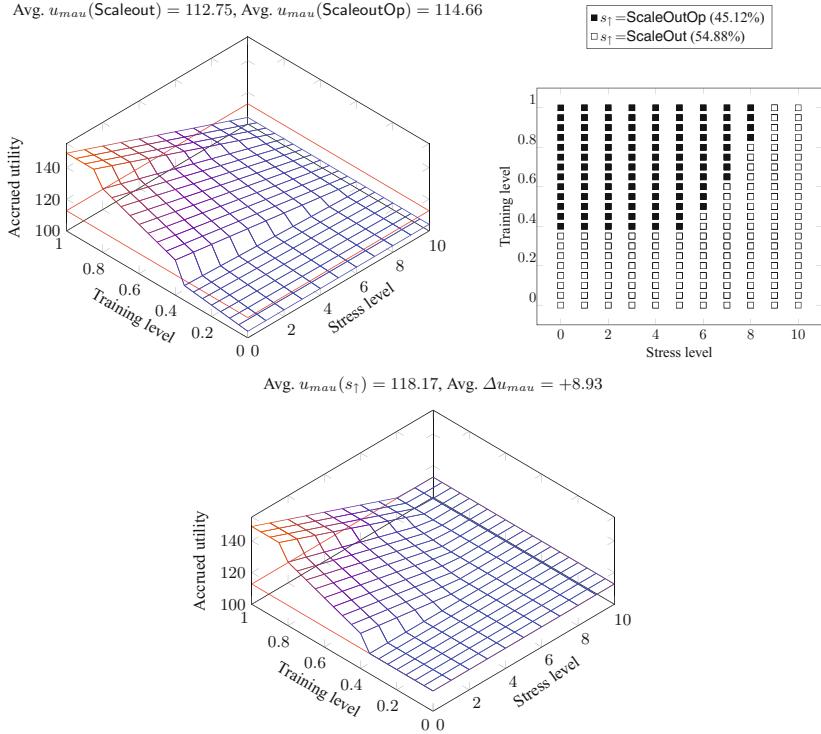


Fig. 6. Experimental results: (a) **ScaleOut** vs **ScaleOutOp** strategy utility (top left), (b) strategy selection (top right), and (c) combined utility (bottom).

The property obtains the maximum accrued utility value (i.e., corresponding to reward rGU) that the system player can achieve until the end of execution ($t = \text{MAX_TIME}$). Figure 6(a) depicts strategy utility analysis results for the different adaptation strategies in a scale out DCAS scenario. In the figure, a discretized region of the state space is projected over the dimensions that correspond to training and stress levels of a human actor (with values in the range [0,1] and [0,10], respectively). Each point on the mesh represents the maximum accrued utility that the system can achieve on a DCAS SMG model instantiated for a time frame [0,200].

If we focus on the curve described by values in the lowest stress level, the figure shows how the use of strategy **scaleOutOp** attains values that go above 140 for maximum training, whereas values below 0.4 are highly penalized and barely yield utility. Moreover, progressively higher stress levels reduce the probability of successful tactic completion, flattening the curve to a point in which training does not make any difference and the strategy yields no utility. On the contrary, the utility obtained by the automated **scaleOut** strategy (represented by the plane in the figure) is always constant since it is not affected by willingness or capability factors.

- *Strategy Selection.* Given a repertoire of adaptation strategies \mathcal{S} , we can analyze their expected outcome in a given situation by computing their expected accrued utility according to the procedure described above. Based on this information, the different strategies can be ranked to select the one that maximizes the expected outcome in terms of utility. Hence the selected strategy s_{\uparrow} can be determined according to:

$$s_{\uparrow} \triangleq \arg \max_{s \in \mathcal{S}} u_{mau}(s)$$

where $u_{mau}(s)$ is the value of property u_{mau} evaluated in a model instantiated with the adaptation logic of strategy s .

Figure 6(b) shows the results of the analysis of strategy selection in DCAS scale out. The states in which human involvement via strategy `scaleOutOp` is chosen ($\approx 45\%$ of states) are represented in black, whereas states in which the fully automated strategy `scaleOut` is selected ($\approx 55\%$) are colored in white. The figure shows that human involvement is only advisable in areas in which the operator has a stress level of 8 and below. Progressively higher stress levels make human involvement preferable only when also progressively higher training levels exist, which is consistent with maximizing the probability of successful adaptation tactic completion. In any case, for training levels below 0.4, human actor participation is not selected even with zero stress level (this is consistent with the function f_C^{addPN} that we define for the capability elements of the human actor, which highly penalizes poorly trained operators). Figure 6(c) shows the combined accrued utility mesh that results from the selection process (i.e., every point in the mesh is computed as $u_{mau}(s_{\uparrow})$). Note that the minimum accrued utility never goes below the achievable utility level of the automatic approach, over which improvements are made in the areas in which the strategy involving human actors is selected. The average improvement in the combined solution corresponds to a percentual improvement of 7.94% over the automatic scale out approach, and 7.81% over the manual one.

5 Conclusions and Future Work

In this chapter, we have described an approach that enables developers to approximate the behavioral envelope of a self-adaptive system by analyzing best- and worst-case scenarios of alternative designs for self-adaptation mechanisms. The approach relies on probabilistic model checking of stochastic multiplayer games (SMGs), and exploits specifications that encode assumptions about the behavior of the environment, which are used as constraints for the generated state-space.

The approach can accommodate different levels of detail in the specification of the environment. On the one hand, rich specifications of environment assumptions can help to reduce the game's state-space and provide a better

approximation of the system's behavioral envelope. On the other hand, an under-specified set of assumptions will result in a larger game state-space due to an over-approximation of the environment's behavior. However, one of the benefits of the approach is that the guarantees given with respect to the satisfaction of the system's objectives can still be relied upon with an under-specified set of assumptions in worst-case scenario analysis. This stems from the fact that the most adverse environment strategy in the over-approximation will always represent a lower bound in the system's guaranteed payoff (either in terms of probability or reward) with respect to any strategies that can be synthesized for more constrained versions of the environment's behavior.

A second advantage of our approach is that it is purely declarative, enabling self-adaptive system developers to obtain a preliminary understanding of adaptation behavior without the need to have any specific adaptation algorithms or infrastructure in place. This represents a reduced upfront investment in terms of effort when compared to other sources of evidence, such as simulation or prototyping.

We have illustrated the versatility of the approach by showing how it can be used to deal with the uncertainty associated with different sources in various contexts, such as self-protecting systems (parameters over time), proactive latency-aware adaptation (simplifying assumptions), and human-in-the-loop adaptation.

A current limitation of the approach is that its scalability is limited by PRISM-games, which currently uses explicit-state data structures and is to the best of our knowledge the only tool supporting model-checking of SMGs. This limitation can be mitigated in some cases by carefully choosing the level of abstraction and relevant aspects of the system and environment in the model. Moreover, we expect that the maturation of this technology will result in the development of symbolic SMG model checkers that will improve scalability.

Regarding future work, we plan to instantiate our adaptation analysis technique in other contexts to deal with uncertainty in cyber-physical and decentralized systems. As regards the application of the technique to human-in-the-loop adaptation, our current models assume that actors and system are working in coalition to achieve goals. In fact, the interaction may be more subtle than that; Eskins and Sanders point out that humans may have their own motivations that run counter to policy [24]. To capture this subtlety, we plan on extending the encoding of SMGs to model human actors as separate players. Moreover, we will extend the human-in-the-loop instance of the approach to formally model and analyze human involvement in other stages of MAPE-K, studying how to best represent human-controlled tactic selection, and human-assisted knowledge acquisition. Concerning latency-aware adaptation, we aim at exploring how tactic latency information can be further exploited to attain better results both in proactive and reactive adaptation (e.g., by parallelizing tactic executions). Finally, we also aim at refining the approach to do runtime synthesis of proactive adaptation strategies.

Acknowledgements. This work is supported in part by awards N000141310401 and N000141310171 from the Office of Naval Research, CNS-0834701 from the National Science Foundation, and by the National Security Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Office of Naval Research or the U.S. government. This material is also based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. (DM-0002292).

References

1. Alur, R., et al.: Alternating-time temporal logic. *J. ACM* **49**(5), 672–713 (2002)
2. Baresi, L., Pasquale, L., Spoletini, P.: Fuzzy goals for requirements-driven adaptation. In: 2010 18th IEEE International Requirements Engineering Conference (RE), pp. 125–134, September 2010
3. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60692-0_70
4. Braberman, V.A., D’Ippolito, N., Piterman, N., Sykes, D., Uchitel, S.: Controller synthesis: from modelling to enactment. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) 35th International Conference on Software Engineering, ICSE 2013, San Francisco, CA, USA, 18–26 May 2013, pp. 1347–1350. IEEE/ACM (2013)
5. Calinescu, R., et al.: Dynamic QoS management and optimization in service-based systems. *IEEE Trans. Software Eng.* **37**(3), 387–409 (2011)
6. Calinescu, R., Kwiatkowska, M.Z.: Using quantitative analysis to implement autonomic IT systems. In: ICSE (2009)
7. Cámara, J., Correia, P., de Lemos, R., Garlan, D., Gomes, P., Schmerl, B.R., Ventura, R.: Evolving an adaptive industrial software system to use architecture-based self-adaptation. In: Litoiu, M., Mylopoulos, J. (eds.) Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2013, San Francisco, CA, USA, 20–21 May 2013, pp. 13–22. IEEE/ACM (2013)
8. Cámara, J., Moreno, G.A., Garlan, D.: Stochastic game analysis and latency awareness for proactive self-adaptation. In: Engels, G., Bencomo, N. (eds.) Proceedings of 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, Hyderabad, India, 2–3 June 2014, pp. 155–164. ACM (2014)
9. Cámara, J., Moreno, G.A., Garlan, D.: Reasoning about human participation in self-adaptive systems. In: Schmerl, B., Inverardi, P. (eds.) Proceedings of 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, Florence, Italy, 18–19 May 2015. ACM (2015)
10. Chen, T., Forejt, V., Kwiatkowska, M., Parker, D., Simaitis, A.: PRISM-games: a model checker for stochastic multi-player games. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 185–191. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_13
11. Chen, T., Forejt, V., Kwiatkowska, M.Z., Parker, D., Simaitis, A.: Automatic verification of competitive stochastic systems. *Formal Methods Syst. Des.* **43**(1), 61–92 (2013)

12. Chen, T., Lu, J.: Probabilistic alternating-time temporal logic and model checking algorithm. In: FSKD, vol. 2 (2007)
13. Cheng, B.H.C., et al.: Software engineering for self-adaptive systems: a research roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_1
14. Cheng, S., Garlan, D., Schmerl, B.R.: Evaluating the effectiveness of the rainbow self-adaptive system. In: 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2009, Vancouver, BC, Canada, 18–19 May 2009, pp. 132–141. IEEE (2009)
15. Cheng, S.-W., Garlan, D.: Stitch: a language for architecture-based self-adaptation. *J. Syst. Softw.* **85**(12), 2860–2875 (2012)
16. Chiulli, R.: Quantitative analysis: an introduction. In: Automation and Production Systems. Taylor & Francis (1999)
17. Deshpande, T., Katsaros, P., Smolka, S., Stoller, S.: Stochastic game-based analysis of the DNS bandwidth amplification attack using probabilistic model checking. In: 2014 Tenth European Dependable Computing Conference (EDCC), pp. 226–237, May 2014
18. Elkhodary, A., Esfahani, N., Malek, S.: FUSION: a framework for engineering self-tuning self-adaptive software systems. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2010, pp. 7–16, New York, NY, USA. ACM (2010)
19. Emami-Taba, M., Amoui, M., Tahvildari, L.: Strategy-aware mitigation using Markov games for dynamic application-layer attacks. In: 2015 IEEE 16th International Symposium on High Assurance Systems Engineering (HASE), pp. 134–141, January 2015
20. Epifani, I., et al.: Model evolution by run-time parameter adaptation. In: ICSE. IEEE CS (2009)
21. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: Atlee, J.M., Inverardi, P. (eds.) Proceedings of 31st International Conference on Software Engineering, ICSE 2009, 16–24 May 2009, Vancouver, Canada, pp. 111–121. IEEE (2009)
22. Esfahani, N., Kouroshfar, E., Malek, S.: Taming uncertainty in self-adaptive software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE 2011, pp. 234–244. ACM (2011)
23. Esfahani, N., Malek, S.: Uncertainty in self-adaptive software systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 214–238. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_9
24. Eskins, D., Sanders, W.H.: The multiple-asymmetric-utility system model: a framework for modeling cyber-human systems. In: Eighth International Conference on Quantitative Evaluation of Systems, QEST 2011, Aachen, Germany, 5–8 September 2011, pp. 233–242. IEEE Computer Society (2011)
25. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D.: Automated verification techniques for probabilistic systems. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 53–113. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21455-4_3
26. Garlan, D., Cheng, S., Huang, A., Schmerl, B.R., Steenkiste, P.: Rainbow: architecture-based self-adaptation with reusable infrastructure. *IEEE Comput.* **37**(10), 46–54 (2004)

27. Goldman, R.P., Musliner, D.J., Krebsbach, K.D.: Managing online self-adaptation in real-time environments. In: Laddaga, R., Shrobe, H., Robertson, P. (eds.) IWSAS 2001. LNCS, vol. 2614, pp. 6–23. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36554-0_2
28. He, K., Zhang, M., He, J., Chen, Y.: Probabilistic model checking of pipe protocol. In: 2015 International Symposium on Theoretical Aspects of Software Engineering (TASE), pp. 135–138, September 2015
29. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv.* **40**(3) (2008)
30. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. *Computer* **36**, 41–50 (2003)
31. Kolmogorov, A.N.: Foundations of the Theory of Probability. Chelsea, New York (1956)
32. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
33. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic games for verification of probabilistic timed automata. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 212–227. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04368-0_17
34. Li, W., Sadigh, D., Sastry, S.S., Seshia, S.A.: Synthesis for human-in-the-loop control systems. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 470–484. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_40
35. Myerson, R.B.: Game Theory: Analysis of Conflict. Harvard University Press, Cambridge (1991)
36. Okhravi, H., Rabe, M., Mayberry, T., Leonard, W., Hobson, T., Bigelow, D., Strelein, W.: Survey of cyber moving target techniques. Technical report 1166, Lincoln Laboratory, Massachusetts Institute of Technology (2013)
37. Schmerl, B., Cámera, J., Gennari, J., Garlan, D., Casanova, P., Moreno, G.A., Glazier, T.J., Barnes, J.M.: Architecture-based self-protection: composing and reasoning about denial-of-service mitigations. In: Proceedings of the 2014 Symposium and Bootcamp on the Science of Security, HotSoS 2014, New York, NY, USA, pp. 2:1–2:12. ACM (2014)
38. Schmerl, B., Camara, J., Gennari, J., Garlan, D., Casanova, P., Moreno, G.A., Glazier, T.J., Barnes, J.M.: Architecture-based self-protection: composing and reasoning about denial-of-service mitigations. In: Symposium and Bootcamp on the Science of Security (HotSoS), Raleigh, USA, 8–9 April 2014
39. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B., Bruel, J.: Relax: incorporating uncertainty into the specification of self-adaptive systems. In: 17th IEEE International Requirements Engineering Conference, RE 2009, pp. 79–88, August 2009
40. Yuan, E., Esfahani, N., Malek, S.: A systematic survey of self-protecting software systems. *ACM Trans. Auton. Adapt. Syst. (TAAS)* **8**(4), 17 (2014)
41. Yuan, E., Malek, S., Schmerl, B., Garlan, D., Gennari, J.: Architecture-based self-protecting software systems. In: Proceedings of the Ninth International ACM Sigsoft Conference on the Quality of Software Architectures (QoSA 2013), 17–21 June 2013
42. Zadeh, L.: Fuzzy sets. *Inf. Control* **8**(3), 338–353 (1965)

43. Zadeh, L.: Fuzzy sets as a basis for a theory of possibility. *Fuzzy Sets Syst.* **100**(Supplement 1), 9–34 (1999)
44. Zhang, X., Lung, C.: Improving software performance and reliability with an architecture-based self-adaptive framework. In: Ahamed, S.I., Bae, D., Cha, S.D., Chang, C.K., Subramanyan, R., Wong, E., Yang, H. (eds.) *Proceedings of the 34th Annual IEEE International Computer Software and Applications Conference, COMPSAC 2010, Seoul, Korea, 19–23 July 2010*, pp. 72–81. IEEE Computer Society (2010)

An Approach for Isolated Testing of Self-Organization Algorithms

Benedikt Eberhardinger^(✉), Gerrit Anders, Hella Seebach, Florian Siefert,
Alexander Knapp, and Wolfgang Reif

Institute for Software and Systems Engineering, University of Augsburg,
Augsburg, Germany

{eberhardinger, anders, seebach, siefert, knapp, reif}@isse.de

Abstract. We provide a systematic approach for testing self-organization (SO) algorithms. The main challenges for such a testing domain are the strongly ramified state space, the possible error masking, the interleaving of mechanisms, and the oracle problem resulting from the main characteristics of SO algorithms: their inherent non-deterministic behavior on the one hand, and their dynamic environment on the other. A key to success for our SO algorithm testing framework is automation, since it is rarely possible to cope with the ramified state space manually. The test automation is based on a model-based testing approach where probabilistic environment profiles are used to derive test cases that are performed and evaluated on isolated SO algorithms. Besides isolation, we are able to achieve representative test results with respect to a specific application. For illustration purposes, we apply the concepts of our framework to partitioning-based SO algorithms and provide an evaluation in the context of an existing smart-grid application.

Keywords: Self-organizing systems · Adaptive systems
Self-organization algorithms · Software engineering
Quality assurance · Software test

1 Introduction

The established quality assurance measure of testing aims at systematically covering possible paths through the system with the goal of finding failures. In this paper we provide a systematic approach for testing self-organization (SO) algorithms. This contribution is an elaboration and extension of the first vision of a framework for testing SO algorithms shown in [12] and is integrated into our overall research road map aiming at testing self-organizing, adaptive systems (SOAS); the vision of this overall approach is outlined in [14].

The properties of the SO algorithms like inherent non-deterministic behavior, an ever-changing environment, a high number of interacting components, and interleaving operations make it hard to achieve a systematic testing approach for SO algorithms. A key to success is automation, since manually it is rarely possible to cope with the high number of demanded test cases (that are necessary as a

result of the huge state space). However, the automation of testing SO algorithms is faced by the complexity of the system class requiring techniques to cope with the following key challenges:

C-ErrorMask. SO algorithms—and in general SOAS—are designed to be robust and flexible under ever-changing environmental conditions. As a side-effect of these properties an SO algorithm itself, other SO algorithms, or adaptation mechanisms might cover the tracks of possible failures that should be revealed during testing the SO algorithms. Thus, faulty behavior of one SO algorithm could be compensated by another mechanism masking the failure.

For instance, assume an erroneous SO algorithm that returns wrong or inappropriate system structures as a result to the controlled components. This fault then could be masked by an adaption mechanism of the components that compensates the wrong system structure by a high and costly amount of adaptation. The SO algorithm would encompass a fault, but no failure is visible at first glance since the adaptation mechanism masks it by keeping the system alive.

C-Isolate. SO algorithms are based on interaction with the system's components. In the majority of cases, several different SO algorithms are incorporated; their overlap of interaction with the components leads to so-called interleaved feedback loops. These are challenging in testing, because it is hard to get dedicated results for a single SO algorithm. To address this challenge, there is a need for isolated testing of single SO algorithms.

As an example of this challenge, assume an SO algorithm that forms organizational structures, e.g., by partitioning the system's components. A further algorithm, however, performs its calculations for parametrization of the components based on this structure, e.g., for forming an evenly distributed output for each partition. These two algorithms are interleaved and a dedicated test result for the algorithm performing its calculation on the structure is only possible if they are isolated, since the results depend on the results of the first and vice versa. However, this isolation is hard to achieve due to the high dependencies between the two algorithms.

C-Oracle. The oracle problem is a well-known challenge for all testing endeavors [9, 23]. However, the properties of SO algorithms increase this problem: For classical testing the conditions of execution for the system under test (SuT) as well as the concrete requirements are known. Let us call these facts the “known-knowns”.¹ For SO algorithms under test (SOuT) we know that there are unknown conditions of execution where we can hardly decide a priori, i.e., at design-time, whether a state is correct or not; we call these conditions the “known-unknowns”. Moreover, for the SOuT there might even be situations we are not aware of at all, which we call the “unknown-unknowns”. An oracle that

¹ The classification of known-knowns, known-unknowns, and unknown-unknowns is borrowed from United States Secretary of Defense Donald Rumsfeld’s response given to a question at a U.S. Department of Defense news briefing on February 12, 2002.

is capable of evaluating the test results of SO algorithm at least has to be able to handle the “known-unknowns”.

For an instance of “known-unknowns” consider a smart energy grid setting (which is outlined in more detail in Sect. 2) where different power plants are self-organized in different so-called autonomous virtual power plants. If weather-dependent power plant are include, the SO here will depend on the weather conditions. We know that there are different conditions like sunny, rainy, and windy, but we also know that we do not know all different possible combinations and the according correct organizational structures at design-time of the test (or at least we cannot compute all). However, an oracle has to cope with that and has to decide whether a result is accepted as correct or rejected as incorrect.

C-BranchingStateSpace. A huge state space is a common challenge for software testing [9], but, as for the oracle problem, SO algorithms add a further dimension. Most of the approaches in software testing coping with a huge state space make use of the structure of the state space to reduce the amount of test cases needed to be executed. For instance, an infinite loop in a code fragment means an infinite state space, but its ramification degree is rather small. A mechanism applied here is boundary-interior-testing [33] that cuts deep branches at certain lengths. SO algorithms, however, are mostly based on heuristics for coping with the ever-changing environment, making the result non-deterministic; these lead to a wide and rather flat branching structure of the state space and make most of the classical techniques hardly applicable directly.

We address the challenges for testing SO algorithms in our framework *Isolated Testing of Self-organization Algorithms* (IsoTeSO) encompassing test case generation, execution, and evaluation. It provides techniques and concepts for methodically decomposing and isolating the SOuT and executing them in a controlled environment. The isolation can take place on several levels, e.g., isolation from other SO algorithms or from the environment controlled by the SO algorithm itself. Addressing C-Isolate further enables to cope with C-ErrorMask since we can reduce disturbances with the testing environment. However, it turns out that this measure is not enough to address C-ErrorMask, since for SO algorithms operating in several phases the error masking may occur within the SO algorithm itself. We hence introduce a gray-box access to the SOuT by the oracle to be able to detect failures hidden in the depth of the SO algorithms. This is embedded in the automated oracle of IsoTeSO whose implementation addresses C-Oracle based on the concept of what we call the corridor of correct behavior. This corridor allows to distinguish between correct and incorrect states even for the “know-unknowns”, making it capable for evaluating the test results of the SO algorithm. Test case generation within IsoTeSO is based on a static and dynamic test model concept that is capable of handling the complex environment of SO algorithms, addressing C-BranchingStateSpace by systematically selecting test cases based on their occurrence probability within the application domain.

The remainder of this paper is structured as follows. Section 2 introduces the smart-grid case study used for evaluation and Sect. 3 presents our main ideas for testing SOAS. In Sect. 4, we detail the building blocks of the IsoTeSO framework

including the test models. The framework is then used for testing two different SO algorithms introduced in Sect. 5. The evaluation in Sect. 6 confirms that our approach encompasses an efficient combination of model-based and random generation techniques based on our test models for finding different kinds of (injected) faults in the examined SO algorithms. Section 7 discusses related work. We conclude with a summary and some ideas for future work in Sect. 8.

2 Case Study: Self-Organized Creation of Virtual Power Plants in Smart Grids

The wide-spread installation of weather-dependent power plants as well as the advent of new consumer types like electric vehicles put a lot of strain on power grids. Additionally, small dispatchable power plants (e.g., biogas plants) owned by individuals or cooperatives feed in power without external control. To save expenses, gain more flexibility, and deal with uncertainties, future *autonomous* power management systems have to take advantage of the full potential of dispatchable prosumers² by incorporating them into the scheduling scheme. Further, uncertainties have to be anticipated when creating schedules and compensated for locally to prevent their propagation through the system.

To meet the challenges of future power management systems, Steghöfer et al. presented the concept of *Autonomous Virtual Power Plants* (AVPPs) in [44] (similar visions of virtual power plants are discussed in [37]). AVPPs represent self-organizing groups of two or more power plants of various types (cf. Fig. 1). The organizational structure represents a *partitioning*, i.e., every power plant is a member of exactly one AVPP, which is established and maintained by a (partitioning-based) SO algorithm. Constraints that specify valid partitionings, e.g., a maximum number of power plants that may belong to one AVPP or that every power plant has to belong to exactly one AVPP, among others, induce a *corridor of correct behavior* over the space of all partitionings. In this setting, each AVPP has to satisfy a fraction of the overall demand. To accomplish this task, each AVPP autonomously and periodically calculates schedules for directly subordinate dispatchable power plants. Further, each AVPP's dispatchable power plants have to reactively compensate for deviations resulting from local output or load fluctuations (i.e., uncertainties) to avoid affecting other parts of the system.

AVPPs autonomously adapt their structure to changing internal or environmental conditions, they are able to live up to the responsibility of maintaining an organizational structure enabling the system to hold the balance between energy supply and demand. In particular, if an AVPP repeatedly cannot satisfy its assigned fraction of the overall demand or compensate for its local uncertainties, it triggers a reorganization of the partitioning. The goal is to form homogeneous partitionings in the sense of a structure of similar AVPPs that are likely to feature a heterogeneous composition: On the one hand, by distributing

² We use the term “prosumer” to refer to producers as well as consumers.

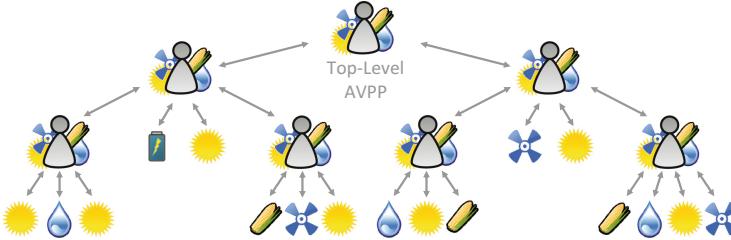


Fig. 1. Hierarchical system structure of a future autonomous and decentralized power management system: Power plants are structured into systems of systems represented by AVPPs that act as intermediaries to decrease the complexity of control and scheduling. AVPPs can be part of other AVPPs. The left child of the top-level AVPP, for instance, controls a solar power plant, a storage battery, and two subordinate AVPPs.

unreliable power plants among AVPPs, the chance of fluctuations is reduced and the system's robustness increases. On the other hand, by balancing the AVPPs' degrees of freedom (e.g., by mixing different generator types), their ability to locally deal with uncertainties, i.e., fluctuations, is promoted. To cope with the vast number of dispatchable power plants, the concept of AVPPs proposes a scalable, hierarchical structure in which AVPPs act as intermediaries. This system decomposition reduces the number of dispatchable power plants (including directly subordinate AVPPs) each AVPP controls resulting in shorter scheduling times for each AVPP and the overall system.

In the smart-grid application, we have to cope with C-Isolate as well as C-ErrorMask since the partitioning algorithm and the mechanism that balances supply and demand in each AVPP influence each other significantly. Error masking can also occur in the partitioning algorithms if, e.g., the result of the SO algorithm is faulty but the system state after the reorganization process is valid. The problem of state space explosion (C-BranchingStateSpace) has to be tackled in this case study due to the stochastic nature of the partitioning algorithms in order to deal with the large search spaces, the non-deterministic behavior of single agents, and the stochastic environment (e.g., weather conditions, market prices). Finally, C-Oracle occurs in this context since it is hardly decidable at design-time which partitioning for which power plants is correct as this depends on the unknown environmental setting of the controlled power plants as well as on the unknown states of the autonomous power plants themselves.

3 The Corridor Enforcing Infrastructure (CEI) for Testing Self-Organizing, Adaptive Systems

Our approach for testing SOAS—and consequently for testing SO algorithms—is based on the *Corridor Enforcing Infrastructure* (CEI) [14]. The CEI is an architectural pattern for SOAS using decentralized feedback-loops to monitor and control single agents or small groups of agents in order to ensure that the

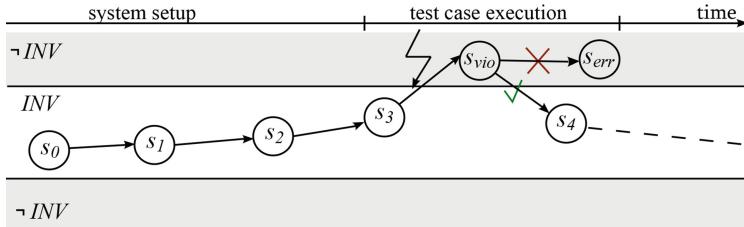


Fig. 2. Schematic state-based view of the corridor of correct behavior and the different phases of testing SO algorithms. INV is the conjunction of all constraints of the system controlled by the CEI.

system's requirements are fulfilled at run-time. Within the CEI the concepts and fundamentals of the *Restore Invariant Approach* (RIA) [22] are applied. RIA defines the *Corridor of Correct Behavior* (CCB), which is described by requirements concerning the system's structure, formalized as constraints. Concerning the smart-grid scenario (introduced in Sect. 2) the CCB is formed by the constraints describing valid partitionings for the AVPPs, e.g., a maximum and minimum number of power plants that are allowed in each AVPP. The conjunction of all these constraints is called the *invariant* (INV). An exemplary corridor is shown in Fig. 2. If INV is satisfied, the system is inside the corridor; otherwise, the system leaves the corridor, indicated by the flash. At this point the system has to be reorganized in order to return into the corridor, as shown by the transition with a check mark. A failure occurs in this context if a transition outside of the corridor is taken, indicated by a cross.

The CEI implements the RIA with decentralized pairs of a monitor and a controller, similar to the MAPE cycle [25] or the Observer/Controller (O/C) architecture [41]. The schematic view in Fig. 3 shows an implementation of the CEI based on the O/C architecture where the essential parts are the system under observation and control (SuOC, i.e., single agents or groups of agents controlled), the observer (O, i.e., the component monitoring the state of the SuOC and providing information to the controller) and the controller (C, i.e., the self-x-algorithms controlling the SuOC, e.g., an SO algorithm). Note that the CEI consists of sets of nested feedback-loops controlling the entire system. Figure 3 shows further the different layers of testing applied to cope with the complexity of the system: agent, interaction, and system layer. The IsoTeSO framework is located in this concept on the *Interaction Layer* where the SO algorithms are incorporated into the *Controller*.

In some cases the observer for single agents or small groups of agents can be generated (semi-)automatically from the requirements documents for the considered system, as shown by Eberhardinger et al. [15]. The small groups controlled in the smart-grid scenario are the power plants partitioned into AVPPs. Furthermore, single power plants are in control loops to adjust, in particular, their energy production. The reorganization by the controller is performed by one or more SO algorithms resulting in a new system configuration. Such a system

configuration has to satisfy the constraints describing valid organizational structures, e.g., a maximum number of controlled components within each organization. With regard to the partitioning problem—applied for the AVPPs—this means that each power plant belongs to exactly one AVPP. For other application scenarios one might require a structure in which each component belongs to at least one organization, which corresponds to a set covering [7]. The concrete choice of the organization algorithms and their constraints has no impact on our approach, both set covering and partitioning SO algorithms could be implemented within the concept of the CEI.

In order to grasp SO algorithms for testing, we need techniques to examine the CEI and its mechanisms, covering the following responsibilities of the CEI (marked with a rounded rectangle in Fig. 3): correct initiation of a reorganization if and only if a constraint is violated (monitoring infrastructure, R-Detect); calculation of correct system configurations in case of violations (R-Solution); and correct distribution of these configurations within the single agents or small groups of agents controlled by the CEI (R-Distribution). At the moment, IsoTeSO focuses on revealing the kinds of SO algorithms failures related to (R-Solution) and (R-Distribution). Section 8 gives an outlook on the possible extensions of the framework for testing the monitoring infrastructure (R-Detect).

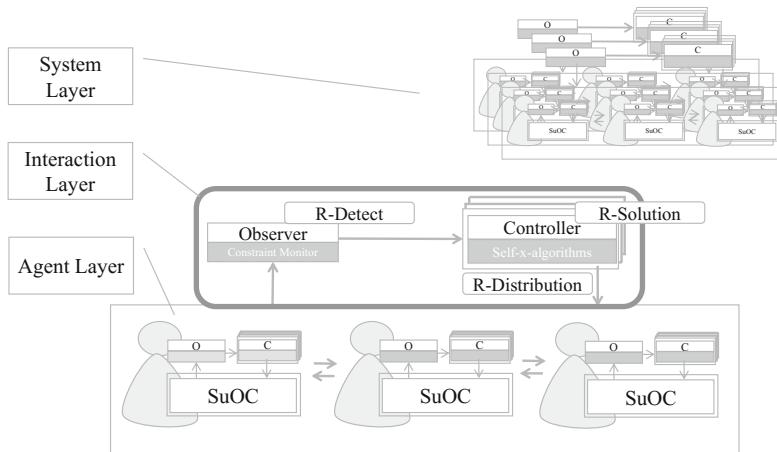


Fig. 3. Schematic view of the *Corridor Enforcing Infrastructure* (CEI) and its different level of testing (namely, agent, interaction, and system layer). The IsoTeSO Framework is located in this concept on the *Interaction Layer* where the SO algorithms are incorporated into the *Controller*. Besides the *Controller* (*C*) the CEI main components are the *System under Observation and Control* (*SuOC*), and the *Controller* (*C*) that form together distributed, decentralized feedback-loops on different levels, i.e., the CEI consists sets of nested feedback-loops (symbolized by the nested set of different system parts in the upper part of the figure) controlling the entire system.

On the other hand, the CCB immediately affords for our testing efforts a check whether a system configuration produced by an SO algorithm for reorganization is correct, i.e., whether all constraints of the CCB hold. This integrated check consequently forms the basis of a test oracle in the IsoTeSO framework. However, in order to use the CCB correctness check for result validation, a consistent snapshot of the system state is needed. Therefore, in a first step, IsoTeSO is built upon a stepwise execution model that naturally allows to synchronize all system components at distinct points in time. The necessary synchronization may reduce the possible interleavings of component behaviors, but this may be mitigated by letting components decide for making an idle local step in every synchronous round. Still, the concept of the CCB makes it possible to create a fully automated oracle that allows to tackle C-Oracle, since it is possible to decide even under unknown conditions of execution whether a state is inside the CCB or not, addressing an evaluation of the “known-unknowns”.

4 A Framework for Isolated Testing of Self-Organization Algorithms (IsoTeSO)

As common for testing frameworks, we organize IsoTeSO in a process-oriented manner into: test case generation³ (cf. Sect. 4.2), test execution (cf. Sect. 4.3), and, finally, test evaluation as a part of output processing (cf. Sect. 4.4). An overview of the framework is shown in Fig. 4.

To derive appropriate test suites within the test suite generator component, we use the test model presented in Sect. 4.1 that enables sufficient abstraction to tackle C-BranchingStateSpace. IsoTeSO allows for isolating SO algorithms from other parts of the system, including other SO algorithms (C-Isolate). For this purpose, IsoTeSO provides a fully controlled test environment, i.e., the environment of the SOuT is fully mocked. To enable mocking, the execution component, described in Sect. 4.3, provides a system simulator and a gray-box interface for the SOuT. The gray-box interface enables the evaluation of the internal state of the SOuT, e.g., for checking interim results, and thus also allows us to address C-ErrorMask. The evaluation of the validity of reconfigured system structures is performed by the test oracle that checks the violation of constraints describing valid solutions (C-Oracle). The responsible component for the test evaluation—the monitoring and evaluation component—is described in Sect. 4.4.

Overall, we assume that the test system is embedded into an agent-based execution system. To allow a consistent snapshot and therefore coherent synchronization points after each executed test case, the system is based on a stepwise execution model.

³ Due to the fully automated evaluation of test cases by the oracle component of IsoTeSO, test case generation reduces to test input generation as no expected output is needed. This concept builds up on Artho et al. [6] also combining run-time verification and test input generation for creating test cases. In the remainder of this paper we use test case generation in the sense of test input generation.

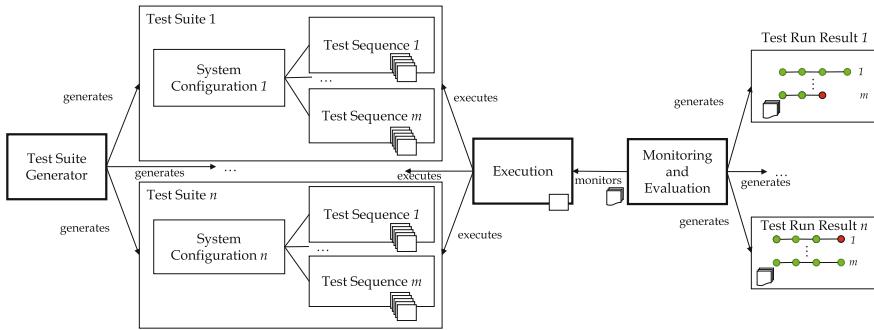


Fig. 4. IsoTeSO generates n test suites, each consisting of an initial system configuration for the test setup and m test sequences. Each test sequence contains a number of test cases (indicated by the overlapping rectangles). Each test sequence is executed individually (indicated by the single rectangle in the execution component). The monitoring and evaluation component monitors the system and samples the reorganization results as well as the system structure for each test case (indicated by the arrow between the monitoring and evaluation component and the execution component). For each test suite, the results of all test cases are evaluated in a test run result, leading to n results for m evaluated test sequences.

4.1 Test Model of the Framework IsoTeSO

The test model of the framework is the source for generating test suites (cf. Sect. 4.2) and is composed of three parts:

1. Model of the system under test (SuT)
2. Model of the SO algorithm under test (SOuT)
3. Model of environmental changes and influences.

The first one provides the necessary information for the domain in which the SOuT is applied, the second one defines suitable configurations for the SOuT which are highly influenced by domain knowledge. The model of environmental changes and influences describes the dynamics of the environment and is mainly used for creating test sequences. In the following, these three parts of the test model are described with respect to their intended purpose within the test suite generation component.

Model of the system under test. The model of the SuT specifies all important information from the domain the SOuT belongs to and is used for generating the system configuration within the test suites. For this purpose, a domain description (in the form of a UML class diagram) is provided that is enriched by constraints describing the CCB (cf. Sect. 3). With regard to our case study, the constraints define valid partitionings, e.g., the maximum and minimum number of power plants for each AVPP as well as the constraint that each power plant must be contained in exactly one AVPP. Further, the behavior of each agent type (e.g., wind turbines, solar panels, or biogas power plants) is defined by standard

design documents. In our smart-grid case study we additionally have to define *groups of agents* which are influenced by similar environmental changes, such as wind turbines with a certain locality. Accordingly, a group of agents share one *environment profile*, a stochastic model abstracting possible environmental changes (more details are given at the end of this subsection). The fact that an agent is in a certain group, is not known to the SOuT and has no direct influence on the partitioning decisions of the SOuT. The members of a group of agents can be of different agent types.

We define how the environment influences certain types of agents so that a reduction of the test cases to realistic scenarios is possible. Indeed, describing the influences relation between the environment and types of agents is rather complex. At this step we rely on simplification and abstraction within the model to keep our approach scalable: We assume that the mapping of environment influences to agent types (in its abstracted form) can be determined and defined a priori, i.e., we neglect that the influence might change over time or is indeterministic. The consequences of this assumption and simplification are on the one hand that the model is scalable within the approach and can be handled by the test engineer, but, on the other hand, that it might neglect some situations for testing. Our evaluation results (cf. Sect. 6.3), however, showed that it is still possible to find different kind of failures.

To recap, the model of the system under test must contain at least:

- Types of agents
- Definition of possible initial states for the agents
- Suitable ranges for the minimum and maximum number of agents of a specific agent type
- Constraints concerning groups of agents (minimum and maximum number as well as size)
- Mapping of environmental influences to agent types
- Constraints concerning valid system structures, i.e., the relevant part of the CCB.

Since the model of the system under test depends on the application domain the description for the model itself is quite generic and coarse. However, the detailed information of the application—given in Sect. 6—shows how to transfer this generic description into a specific model of the system under test.

Model of the SO algorithm under test. This model specifies valid configurations for the SOuT. It is used to derive valid ranges for all relevant parameters, such as the algorithm’s maximum run-time. The selection of relevant parameters highly depends on the concrete algorithm (e.g., some algorithms allow to specify a maximum execution time and some do not), the situations that should be covered by the test runs (e.g., some failures only occur if we give the algorithm enough time), and the domain knowledge (e.g., in some domains, the maximum run-time is naturally bounded). Clearly, despite the random testing approach, a suitable parametrization can be used for directed testing. This means that we can push

the algorithm into interesting directions, e.g., to use specific functionality, which increases the code coverage.

The parameters of the SOuT and dependencies between the parameters are specified as constraints of the permitted setting of the parameters of the SOuT. The resulting constraint satisfaction problem (CSP) is solved by the input component of the framework for forming a system configuration within a test run. For a SOuT that is based on a particle swarm optimization algorithm to form organizational structures (like the PSOPP algorithm described in Sect. 5.2) a parameter of the algorithm is the number of particles to use, constrained in the model of the SOuT, for instance, by an upper and lower bound. Further, the number of particles are related to the constraints concerning the number of partitions to be formed. This relationship is incorporated into the CSP. The described CSP is the foundation for automatically generating valid settings for the SOuT. A more detailed example for the specification is given in Sect. 6.

Model of environmental changes and influences. Recalling our idea of isolated testing of SO algorithms, the third component of the test model is the model of environmental changes and influences. This is because of possible interferences with other SO algorithms due to interleaved feedback loops (C-Isolate) as well as the huge state space induced by the different possible states of the environment and the algorithm's non-deterministic behavior (C-BranchingStateSpace). We address C-BranchingStateSpace by providing stochastic models of the environment, called *environment profiles* (EPs), and C-Isolate by *functions describing the environment's influence* on the system that enables to decouple the SO algorithm.⁴

As we do not assume that all agents share the same environment (e.g., because of their geographical distribution) and are equally influenced by environmental conditions, we define these EPs and influence functions with regard to a specific group of agents \mathcal{G} .⁵ This allows us to deal with large state spaces even better (C-BranchingStateSpace). For example, it is not necessary to consider the complete set of possible states of the environment $\mathcal{E} = \{\text{(cloudy, high price)}, \text{(rainy, high price)}, \text{(sunny, high price)}, \text{(cloudy, low price)}, \text{(rainy, low price)}, \text{(sunny, low price)}\}$ if we regard a group of solar power plants whose output mainly depends on the current weather conditions and is more or less independent of the current market price (a property that is defined in the mapping of environmental influences to agent types). Instead, for each group of agents \mathcal{G} , we map one or more states of the environment from the set \mathcal{E} to a single so-called *relevant state* that describes the relevant parts of the environment's state for \mathcal{G} , i.e., those that have an influence on \mathcal{G} 's behavior. By gathering all relevant states, we obtain the entire set of relevant states $\mathcal{R}_{\mathcal{G}}$ for \mathcal{G} .

⁴ Note that the environment also covers in this case other SO algorithms of the system.

⁵ That technique of state reduction is performed according to the state abstraction principles that are well known in classical testing [33].

In case of our group of solar power plants the states (cloudy, high price), (cloudy, low price) $\in \mathcal{E}$ are mapped to a state **cloudy** that becomes a member of \mathcal{R}_G . In this example, \mathcal{R}_G is finally equivalent to the set of weather conditions {cloudy, rainy, sunny} considered in \mathcal{E} .

The identification of relevant states is supported by the mapping of environmental influences to agent types as described in the model of the system under test. Thus, if the environment state or a set of environment states corresponds to an environmental influence that is already mapped to an agent type of the concerning agent group, this state has to be included into the set of relevant states for this agent group. However, mapping the environment's states to relevant states of an agent group is—as the mapping of environment influences to agent types in the model of the system under test—not generically solvable; in every application domain this classification of relevance has to be made specifically. Further, the relevant states are not limited to the mapping described for the environmental influences to agent types, since, among other things, the other SO algorithms are here also part of the environment (where as that is not considered by the model of the system under test).

The exemplified description above shows what the necessary steps are and how this mapping should be achieved. The mapping in general does not have to be disjoint but we expect it to be complete since only useful environment states should be included in \mathcal{E} , i.e., states that influence at least one of the agent groups.

With regard to a specific group of agents, an EP not only captures the relevant states \mathcal{R}_G of \mathcal{G} 's environment, but also probabilities for changes from one state to another. Assuming that the next state only depends on the current state, an EP represents a first-order Markov chain. Figure 5 depicts a simplified example of an EP for a group of solar power plants in a specific region; as a matter of fact, the models used for testing are much more complex (cf. Sect. 6). Such an EP can either be created using domain knowledge, derived from statistical data gathered during the execution of the system under test, or a combination of both. In the literature of multi-agent systems, Markov chains are often used to simulate the environment for evaluation purposes (cf. [1, 42]).

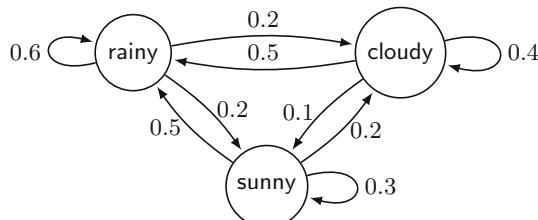


Fig. 5. A simplified EP for a group of solar power plants at a specific geographic location: Possible weather changes between **rainy**, **sunny**, and **cloudy** are indicated by directed edges. The numbers represent transition probabilities.

To model the way the environment influences the members of an agent group \mathcal{G} , we use a function $f_{\mathcal{G}} : \mathcal{R}_{\mathcal{G}} \times \mathcal{S}_{\mathcal{G}} \rightarrow \mathcal{S}_{\mathcal{G}}$, where $\mathcal{S}_{\mathcal{G}}$ represents all possible states of \mathcal{G} 's members. With regard to a member $a \in \mathcal{G}$, the function $f_{\mathcal{G}}$ maps the new state $\sigma'_{env} \in \mathcal{R}_{\mathcal{G}}$ of \mathcal{G} 's environment and a 's current state $\sigma_a \in \mathcal{S}_{\mathcal{G}}$ to a new state $\sigma'_a \in \mathcal{S}_{\mathcal{G}}$. For instance, the change of the current weather conditions from **sunny** to $\sigma'_{env} = \text{rainy}$ could impair a solar power plant's ability to make adequate predictions of its future output, which is reflected in the transition from $\sigma_a = \text{good predictions}$ to $\sigma'_a = \text{bad predictions}$. Different influences of the weather on different types of weather-dependent power plants—represented as different agent groups—can be formalized by group-specific functions $f_{\mathcal{G}}$. On the other hand, if an EP describes possible developments of the prices at an energy market, $f_{\mathcal{G}}$ can model the way a power plant or consumer behaves at the market, i.e., its strategy. For example, if the market price falls below a certain threshold, some consumers might change their strategy to “buy energy”, whereas the producers might become more reluctant to sell their production. As is the case with the creation of EPs, such influence functions can be deduced from domain knowledge and statistical data. Note that, in some cases, the influence function might depend on random variables and thus becomes probabilistic. This reflects the fact that we cannot assume perfect knowledge about the influences of the environment on the agents due to the high complexity and the agents' autonomy. However, it depends on the test designer's choice as well as on the application domain whether to use a probabilistic functions to map the environment influences to the states of agent groups. The drawback of modeling these functions on random variables is that the process of test case generation is less controllable, but it might reveal some interesting test cases. In our evaluation (cf. Sect. 6), we used deterministic functions which especially payed off in the process of getting more control over the test case generation procedure and gaining higher failure detection rates. Of course, there might be applications where detailing the model applies better in order to gain a higher failure detection rate due to the more accurate model that is for instance able to reveal border cases that are failure prone. The choice depends here on the test designer that adapts the presented approach to specific algorithms and application domains.

As we will explain in the next subsection, the EPs are used for the test case generation by simulating the Markov chains, yielding random but representative test sequences describing environmental changes for the different groups of agents. The probabilistic information of the EPs can be used to estimate the relevance of generated test cases and test sequences and their likelihood of occurrence in a realistic setting (C-BranchingStateSpace). As we will see in our evaluation in Sect. 6, this property makes EPs also an important tool for coverage analysis.

4.2 Test Suite Generator Component of IsoTeSO

IsoTeSO's test suite generator component generates different test suites each containing *one* initial system configuration on which basis it randomly creates

n test sequences by using the environment profiles (cf. Fig. 4). A randomly generated initial system configuration contains the following information:

- a set of agents, possibly of different types
- a set of initial agent states, one for each agent
- a set of agent groups so that each agent is contained in exactly one group
- an initial system structure
- the selected SOuT
- additional system parameters, including the parametrization of SOuT and a random seed to be used during the execution of the test sequences.

The fact that all sequences within one test suite start with the same system configuration is important in the context of SOAS to tackle the already mentioned four challenges (cf. Sect. 1). Each test sequence is an ordered series of test cases whose length corresponds to the number of discrete (time) steps that should be simulated in the course of the test run. This is possible since IsoTeSO uses a stepwise execution model. To generate the test cases, a chosen set of EPs is simulated for the specified number of time steps to obtain the sequence of environmental changes.

IsoTeSO’s test suite generator component can be used in two different modes concerning the execution of the framework:

- *Online*: This mode produces new test suites during the execution of the framework. It accordingly enables an “endless” execution of the framework. For this purpose a new test suite is generated and executed after every completed test run.
- *Offline*: This mode assumes a given set of test suites which is generated *a priori* and sequentially executed by the framework.

In both modes, the test suites are executed by the execution component.

4.3 Execution Component of IsoTeSO

The execution component defines two phases: (1) setting up the system and (2) executing the test cases on the SOuT.

Setting up the system. First, the *System Initializer* sub-component processes the initial system configuration provided by the current test suite. For this purpose, it sets up the set of predefined agents, using the given initial agent states, and establishes the given initial system structure. Next the SOuT is incorporated via an interface (**IController**) which plugs in the SOuT via an adapter component. At this stage, we can use IsoTeSO only for black-box testing, since we have no access to interim results of the SOuT. For the isolated testing of our partitioning algorithms, we plugged in SPADA and PSO as shown in Fig. 6. The adapter component is an implementation of the adapter pattern that enables efficient testing of different SO algorithms. Therefore, the adapter fulfills the following tasks, when reorganizing the system structure: (i) it instantiates the specified

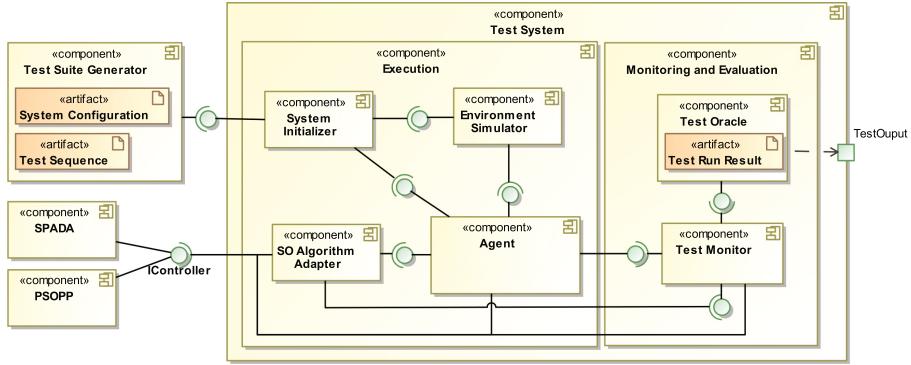


Fig. 6. UML component diagram of the detailed architecture of IsoTeSO having two different partitioning-based SO algorithms plugged in (SPADA, PSOPP). The *Test Suite Generator* component provides the artifacts of a test suite. The *System Initializer* component initializes the system using the generated test suites and sets up the *Environment Simulator* and the *Agents*. Within the *Execution* component, the test suites are executed via the *Environment Simulator* while the *Monitoring and Evaluation* component logs the execution (*Test Monitor*) and the *Test Oracle* evaluates it to *Test Run Results*.

SOuT, (ii) sends the current system structure to this SOuT, (iii) as soon as the SOuT terminates, the adapter informs the test monitor (as described in Sect. 4.4) about the solution, and (iv) after the monitor checked the solution, the adapter requests the SOuT to adopt the new system structure. Point (iii) yields a gray-box view for testing, since we are now able to check interim steps of the test case execution. This is needed for testing if the SOuT—as part of the controller of the CEI—fulfills (R-Solution) besides (R-Distribution).

Executing test cases on the SOuT. After the setup is completed, the environment simulator sequentially executes the given test cases. This execution is based on a stepwise execution model that enable consistent synchronization points after every test case. First, the agent components perceive the corresponding environmental changes and adapt their internal states according to their influence functions f_g . After the changes are applied to every component, the system constraints are checked for a violation, i.e., whether the system leaves the CCB, if so the SOuT is activated in next step via the adapter component.⁶ Based on the current system state, the SOuT has to reorganize the system structure in order to restore the system constraints, i.e., to bring the system back into the CCB. That last action completes one execution sequence that correspond to the executed test case.

⁶ Note that not every test case execution leads directly to constraint violations and thus to an activation of the SOuT. To form a realistic system structure within the test system, it is necessary to allow the system to take transitions that do not violate the CCB.

The gray-box view on the SOuT allows us to tackle (R-Solution) as well as C-Oracle since we can evaluate the calculated system structure before it is distributed among the agents. In addition, the gray-box view consequently avoids error masking during testing—defined as challenge C-ErrorMask. The adapter continuously informs the test monitor about the current system state during the test case execution at specific points in time. Gaining the synchronized system state is possible due to the stepwise execution model. Based on this information, the test oracle can decide if the distribution of the solution leads to a correct system configuration, thereby addressing (R-Distribution).

4.4 Monitoring and Evaluation Component of IsoTeSO

The monitoring and evaluation component of IsoTeSO—that mainly tackles C-Oracle by applying the concepts of the CCB to monitoring and evaluation—is split into two sub-components, each being responsible for a specific task:

1. the test monitor observes and samples data of the executed SOuT and of the test system
2. the test oracle evaluates the sampled data.

To live up to these responsibilities, the monitoring and evaluation component directly interacts with the execution component and the SOuT using the provided interfaces (cf. Fig. 6).

Monitoring and sampling data. In the course of the execution of a test suite, the monitoring and evaluation component monitors and samples data of the test system influenced by the SOuT as well as directly of the SOuT. This allows to observe the two steps of a SOuT: (1) computing a solution for the reorganization, and (2) distributing the solution to the agents. The result of these steps is stored in the current system state, and sent to the test oracle for further evaluation. A necessary prerequisite for monitoring and sampling data is the established stepwise execution model, allowing to synchronize all system components at distinct points in time for later evaluation. However, the synchronization of all system components does not imply prerequisites or restrictions to the SOuT’s characteristic. Having a synchronization point and a global view within the test system does not interfere the decentralization of the SOuT since the SOuT itself makes no use that global view, it is only introduced for test oracle. Indeed, there are some restrictions concerning the possible tested situations. The stepwise execution model neglects test cases that address situations where components are in different states. Incorporating them into our test approach is a matter of future work outlined in Sect. 8.

Evaluating the data by a test oracle. Having received the sampled state for a test case from the test monitor, the constraint-based test oracle evaluates the result of the test case either to *passed* or *failed*. The evaluation of the results is based on constraints, which represent the requirements for the SOAS as well

as the SOuT and form the CCB (as described in Sect. 3). Each constraint must be satisfied by the given system structure to pass the test. The evaluation of the constraints can be automatized by parsing them to checks whether or not a test succeeds. Besides this evaluation based on the constraints, IsoTeSO also supports smoke tests, e.g., observations whether the system throws exceptions during execution or simply crashes. This enables a completely automatized test process within the framework and addresses C-Oracle.

As shown in Fig. 4, the monitoring and evaluation component generates test run results for a specific test suite. The test run result contains evaluations of each test case result (failed, passed) as well as the logged information (system configuration, SOuT solution) during the test run for each executed test case.

5 Tested Self-Organization Algorithms

In this section, we present two SO algorithms, a decentralized approach (Sect. 5.1) and a metaheuristic (Sect. 5.2). The aim of both algorithms is to partition a set of agents $\mathcal{A} = \{a_1, \dots, a_n\}$ into pairwise disjoint subsets, i.e., partitions, that together constitute a *partitioning* at as minimal costs as possible. With regard to our case study, each AVPP represents a partition and the set of all AVPPs corresponds to a partitioning. As SPADA and PSOPP are part of the controller of the CEI, the oracle has to evaluate if their interim results (R-Solution) as well as the resulting system structures (R-Distribution) meet the functional requirements, i.e., satisfy the properties of partitionings (cf. Sect. 6).

5.1 A Decentralized Algorithm for Partitioning Multi-agent Systems

SPADA [4], the *Set Partitioning Algorithm for Distributed Agents*, solves the *complete set partitioning problem* (CSPP) in a general, decentralized manner. In the following, we give a short summary of SPADA's basic functionality and characteristics. A more detailed description can be found in [4].

In SPADA, the agents use an internal graph-based representation of the current partitioning, called *acquaintances graph*, to solve the CSPP. All operations the agents apply to establish a suitable partitioning can therefore be mapped to graph operations. The nodes of the acquaintances graph are the agents participating in the reorganization. Directed edges represent acquaintance relationships between agents. Together the acquaintances form an overlay network that restricts communication to acquainted agents, thereby lowering complexity in large systems. To indicate that an agent is not only acquainted with another but also in the same partition, edges can be marked. Partitions are thus defined by the transitive-reflexive closure of the binary relation given by the marked edges. Each partition has a designated leader that is responsible for optimizing its composition according to application-specific criteria. An example of such an acquaintances graph is depicted in Fig. 7 (more details concerning the acquaintances graph can be found in [4]).

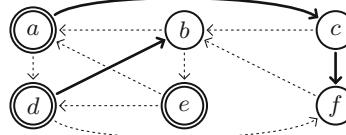


Fig. 7. An exemplary acquaintances graph for a system consisting of six agents (cf. [4]): Agents are represented as nodes and acquaintances as directed edges, e.g., d is acquainted with b and f . Marked edges (symbolized as solid arcs) indicate that their tail and head belong to the same partition. In this example, there are three partitions $\{a, c, f\}$, $\{b, d\}$, $\{e\}$ with leaders a, d, e .

To improve its partition, each leader periodically evaluates if it is beneficial to integrate new agents or to exclude some of its members (e.g., with regard to our case-study, to improve the equal distribution of unreliable power plants among AVPPs). The latter can be beneficial in case of reorganizations that require to create new partitions, e.g., if a partition's or an agent's properties have changed so that the partition's formation criteria no longer favor including the agent. The integration and exclusion of agents is implemented by modifying the edges in the acquaintances graph.

To decide about termination, leaders periodically evaluate application-specific termination criteria. These are formulated as constraints which can also be monitored at run-time to trigger reorganization. If the termination criteria are met, the leader marks its partition as terminated. As long as a partition is marked as terminated, its leader does not change its structure. However, the termination labeling is removed if the partition is changed from outside, i.e., if one of its members is integrated into another partition. This characteristic allows SPADA to make selective changes to an existing partitioning, which is very useful in dynamic environments. It has been shown empirically that SPADA's local decisions lead to a partitioning whose quality is within 10% of the optimum [4].

With regard to our case study, each leader instantiates a new AVPP agent as soon as all partitions terminated. The AVPP then assumes control of all power plants in the partition. In case of a reorganization, the acquaintances graph is created on the basis of the existing system structure.

5.2 A Particle Swarm Optimizer for Partitioning Multi-agent Systems

PSOPP [3], the *Particle Swarm Optimizer for the Partitioning Problem*, is based on *Particle Swarm Optimization* (PSO) [24], a bio-inspired computational method and metaheuristic for optimization in large search spaces. In PSO, a number of particles concurrently explore the search space in search of better candidate solutions by modifying their current positions (at random or by approaching other candidate solutions) as long as a specific termination criterion is not

met. During this process, each particle's current position represents a specific candidate solution. To be able to improve the quality of candidate solutions in a target-oriented manner, each particle Π_i is aware of its best found solution \mathcal{B}_i and the best found solution $\mathcal{B}_{\mathcal{N}_i}$ in its neighborhood \mathcal{N}_i . The algorithm's outcome is the global best found solution \mathcal{B} .

PSOPP solves a variant of the CSPP in the presence of *partitioning constraints* that constrain feasible partitions in terms of a minimum s_{min} and a maximum s_{max} size as well as a minimum n_{min} and a maximum n_{max} number of partitions. Therefore, the test oracle additionally has to check—without interfering the algorithm, by evaluating the logged data of the algorithm and not locking it during execution—if the interim results and the resulting system structure satisfy the partitioning constraints. In PSOPP, each particle represents a partitioning that satisfies the partitioning constraints. The central idea—which could also be applied to other metaheuristics—is to allow the particles to move around the search space by using the basic set operations *join*, *split*, and *exchange*. The join operation creates the union of two partitions, the split operation divides an existing partition into two non-empty subsets, and the exchange operation exchanges elements between two partitions. Particles can apply these operations at random as well as in a target-oriented manner. The main purpose of the former case is to enable the particles to explore the search space by randomly modifying their represented partitioning, i.e., their position. The latter case, in contrast, allows particles to exploit existing candidate solutions by approaching other candidate solutions in promising regions of the search space. Since PSOPP's operations are defined in a way that their application always maintains solution correctness, it combs through a search space that only contains correct solutions. This is advantageous with regard to its performance. Because PSOPP is initialized with a correct candidate solution, it is an any time algorithm.

Similar to SPADA, PSOPP can be customized to a specific application by devising an appropriate fitness function that assesses the quality of solutions and thus steers the search for them. Due to these characteristics, PSOPP can be applied to many different applications in which solving the partitioning problem considered in this paper is relevant and global knowledge is available.

Having specified valid partitionings by means of $n_{min}, n_{max}, s_{min}, s_{max}$ as well as the particles' attitude towards exploration and exploitation by fixing some parameters that influence the probability that particles make a random move or approach other candidate solutions, PSOPP creates a predefined number of particles at random or predetermined positions in the search space (the set of particles does not change at run-time). The latter is suitable when a reorganization of an existing system structure has to take place: If the current structure does not contradict the partitioning constraints, it can be used as a starting point for the self-organization process. Mixing predefined and randomly generated initial partitionings allows to hold up diversity. When searching for an initial system structure, particles are created at random positions.



Fig. 8. Actions performed by particles in each iteration [3].

As long as a specific termination criterion is not met, a particle Π_i performs the following actions in each iteration; these are also depicted in Fig. 8:

1. Evaluate the fitness $f(\mathcal{P})$ of the represented partitioning \mathcal{P} .
2. If the particle's fitness $f(\mathcal{P})$ is higher than the fitness $f(\mathcal{B}_i)$ of its best found solution \mathcal{B}_i , set \mathcal{B}_i to \mathcal{P} . Further, inform all other particles Π_j that contain Π_i in their neighborhood \mathcal{N}_j about the improvement so that they can update $\mathcal{B}_{\mathcal{N}_j}$, i.e., the best found solution in their neighborhood.
3. Update the best found solution $\mathcal{B}_{\mathcal{N}_i}$ in Π_i 's neighborhood \mathcal{N}_i .
4. Stop if the termination criterion is met.
5. Otherwise, randomly opt for the direction in which to move, i.e., choose whether a random move or an approach operation should be applied. In case of an approach operation, also determine the position (i.e., \mathcal{B}_i or $\mathcal{B}_{\mathcal{N}_i}$) that should be approached.
6. Determine the new position \mathcal{P}' by applying the selected move operation to \mathcal{P} .

Once all particles terminated, PSOPP returns the best found solution \mathcal{B} . Possible termination criteria are, e.g., a predefined amount of time, a predefined number of iterations (i.e., moves through the search space), a predefined threshold for the minimum fitness value, or a combination of these criteria.

6 Evaluation

In our evaluation, we used a Java implementation of IsoTeSO that is based on a multi-agent system called TEMAS [2]. We opted for TEMAS because it supports a stepwise execution out of the box, which allows IsoTeSO to monitor consistent states of the system at specific points in time.

As foundation of the *model of the system under test*, we used the domain model of the AVPP application, which can be found in [5]. The model of the environmental changes and influences described the effect of different environmental conditions, such as weather conditions, on the power plants' ability to make adequate predictions of their future output. Clearly, this effect depends on the concrete type of agent, i.e., power plant. For this reason, we regarded four different agent types: solar panels, wind turbines, biogas power plants, and hydro power plants. Based on the assumption that the effect of changing environmental conditions, such as the global radiation, the wind speed, the available amount of

biogas, or the water flow, is characteristic of a specific type of power plant, we generated different sets of agent groups. Further, the power plants' geographic location was taken into account. Considering an AVPP's prediction accuracy as a property resulting from the average prediction accuracy of its members, the system's goal was to maintain a structure of AVPPs that feature a similar prediction accuracy (cf. Sect. 2). As soon as the dissimilarity of the AVPPs' prediction accuracy exceeded a certain threshold (as a result of environmental changes), the power plants triggered a reorganization that should reestablish the similarity. We parametrized the *model of the system under test* as follows:

- number of agents $\#ag$: between 2 and 1000
- agent group size: between 2 and $\#ag$
- number of agent groups: between 1 and $\lfloor \frac{1}{2} \cdot \#ag \rfloor$
- partition size: $2 \leq s_{min} \leq s_{max} \leq \#ag$
- number of partitions: $1 \leq n_{min} \leq n_{max} \leq \lfloor \frac{1}{2} \cdot \#ag \rfloor$
- 10 test sequences per test suite
- number of test cases per test sequence: between 50 and 1000
- number of states per EP: between 3 and 25

As SOuT, we integrated a Java-based implementation of the partitioning algorithms SPADA and PSOPP (cf. Sect. 5). Both implement the IController interface that is used by the SO Algorithm Adapter to initiate the SOuT, request results, and ask the SOuT to adopt the new system structure (cf. Fig. 6). The latter was implemented by sequentially moving the power plants contained in the calculated partitioning from their current AVPP into the corresponding new AVPP. With regard to the system structure, this procedure assures that every power plant is always contained in exactly one AVPP. If the last power plant was removed from an AVPP, this AVPP was dissolved. The description of the algorithms provided in [3, 4] as well as their implementation served as the *model of the SOuT*. As explained in Sect. 4.1, we used this information to identify relevant parameters and suitable valid ranges for their parametrization. For SPADA and PSOPP, we identified the following parameters:

- SPADA
 - number of acquaintances per agent: between 1 and 20
 - number of agents each leader evaluates for integration into its partition: between 1 and 10
 - maximum number of agents a leader can integrate into its partition within a single step: between 1 and 10.
- PSOPP
 - number of particles $\#P$: between 1 and 4
 - number of particles starting at the current partitioning: between 0 and $\#P$
 - probabilities $c_{rdm}, c_{\mathcal{B}_i}, c_{\mathcal{B}} \in [0, 1]$ (with $c_{rdm} + c_{\mathcal{B}_i} + c_{\mathcal{B}} = 1$) to apply a random move operator, approach the particle's best found solution \mathcal{B}_i , and approach the global best found solution \mathcal{B} , respectively
 - max. run-time in seconds: between 1 and 10.

After each reorganization, the oracle checks if the algorithm’s result complies with the definition of partitionings (i.e., each power plant must be a member of exactly one AVPP). As explained in Sects. 4.3 and 4.4, these checks are performed once the algorithm indicates its termination (R-Solution) as well as after the result has been adopted (R-Distribution). Only in case of PSOPP, the oracle additionally evaluated the satisfaction of the partitioning constraints introduced in Sect. 5.2 since SPADA does not allow to restrict valid partitionings with regard to the size and number of partitions.

6.1 Fault Injection

To evaluate our approach, we injected four faults into the SPADA and five faults into the PSOPP implementation. According to Püschel et al. [36], all these faults can be assigned to the classes of “permanent and transient RECONF faults”. Our injected faults have in common that they do not cause the algorithm to throw an exception that simply has to be caught by the oracle (i.e., smoke tests), but that their application can result in an invalid reorganization result or invalid system structure. In a preliminary evaluation that ran for about one week, we tested the SPADA and PSOPP implementation without injecting any faults. We did not observe any failures in the course of these tests. So we can be confident that the failures the oracle reported during our subsequent evaluation can be attributed to a specific injected fault.

SPADA: Injected Faults. The first two types of SPADA faults (cf. SPADA-F1 and SPADA-F2) manifest in an incorrect transformation of the current system structure into SPADA’s internal model of a partitioning, i.e., the acquaintances graph (cf. Sect. 5.1). This false mapping results in an invalid reorganization result.

SPADA-F1/SPADA-F2

- *Description:* When creating the acquaintances graph for a new reorganization on the basis of the current system structure, an arbitrary AVPP is not represented in the acquaintances graph if the number of AVPPs is above (in case of SPADA-F1) or below (in case of SPADA-F2) a certain threshold. We set these thresholds to 100 for SPADA-F1 and to 5 for SPADA-F2.
- *Effect:* The resulting partitioning does not contain the power plants that have been members of the “forgotten” AVPP.

The two other types of faults we integrated into SPADA concern a functionality that is used to transform the result, given in the form of an acquaintances graph, into a set of sets. This functionality is used to provide the result to the SO Algorithm Adapter (R-Solution) and as a preprocessing step to create the new AVPP structure (R-Distribution).

SPADA-F3

- *Description:* In case the size of a partition exceeds a predefined threshold, arbitrary power plants are deleted from this partition until its size equals this threshold. In our evaluation, we set this threshold to 100.
- *Effect:* Some power plants are not represented in the partitioning.

SPADA-F4

- *Description:* In case the size of a partition exceeds a predefined threshold, this partition is replaced by a partition that is randomly selected from the partitioning. In our evaluation, we set this threshold to 100.
- *Effect:* Some power plants are not represented in the partitioning, whereas others occur two or more times.

All SPADA faults can be detected using the gray-box interface (R-Solution), i.e., before the underlying system structure is changed. Given the way the result is transformed into a new system structure (it is ensured that every power plant is always a member of exactly one AVPP), these faults *cannot* be detected using the black-box view (R-Distribution). Note that the oracle does not check the system structure with respect to the number and the size of AVPPs in case of SPADA. For each type of injected fault, we are thus confronted with the problem of error masking (C-ErrorMask).

PSOPP: Injected Faults. Regarding PSOPP, we modified the implementation of the move operations “random split” (PSOPP-F1), “random join” (PSOPP-F2), “approach split” (PSOPP-F3), “approach join” (PSOPP-F4), and “approach exchange” (PSOPP-F5) as described in the following listing.

PSOPP-F1

- *Description:* If a partition K is randomly split into two partitions L and M , an arbitrary power plant of L is replaced by another arbitrary power plant of M . This fault does only occur if the size of L and M is below a threshold t_1 or above a threshold t_2 .
- *Effect:* With regard to the resulting partitioning, a specific power plant is missing and another occurs twice.

PSOPP-F2

- *Description:* If two partitions K and L are merged into a new partition M when applying the random join operator, either K or L is not removed from the partitioning. This fault does only occur if the size of K and L is below t_1 or above t_2 .
- *Effect:* In the resulting partitioning, the power plants of either K or L occur twice as they are also contained in M .

PSOPP-F3

- *Description:* If a partition K is split into two partitions L and M , the resulting partitioning does not contain either L or M . This fault does only occur if the size of L and M is below t_1 or above t_2 .
- *Effect:* In the resulting partitioning, the power plants of either partition L or M are missing.

PSOPP-F4

- *Description:* If two partitions K and L are merged into a new partition M when applying the approach join operator, one element is removed from M . This fault does only occur if the size of K and L is below t_1 or above t_2 .
- *Effect:* In the resulting partitioning, a single power plant is missing.

PSOPP-F5

- *Description:* If some power plants are exchanged between two partitions K and L , one power plant of either K or L occurs in both resulting partitions M and N . This fault does only occur if the size of M and N is below t_1 or above t_2 .
- *Effect:* In the resulting partitioning, one power plants occurs twice.

In our experiments, we used $t_1 = 2$ and $t_2 = 100$ so that the failures do only occur in certain situations. Note that the application of an injected fault does not necessarily yield an invalid result because an invalid candidate solution must be rated better than all other (possibly valid) candidate solutions found by the particles (C-ErrorMask). Clearly, this characteristic together with PSOPP’s non-deterministic behavior exacerbates the detection of an injected fault (C-BranchingStateSpace).

In principle, all types of PSOPP faults could lead to a false result that can be detected using the gray-box interface (R-Solution) as well as the black-box view (R-Distribution). Note, however, that not all invalid results manifest themselves in an invalid system structure. Consider the following example illustrating error masking (C-ErrorMask): Assume that the power plants a and b are currently members of the same AVPP. If a reorganization causes an invalid result that does not contain these two power plants, the oracle detects a failure using the gray-box interface. However, the resulting system structure is *valid* in case the minimum size of an AVPP is ≤ 2 and the maximum number of AVPPs is not exceeded. This is because a and b simply remain in their old AVPP if they are not contained in the provided result.

6.2 Test Execution

To be able to make a clear statement which types of faults can be found by IsoTeSO, only one specific type was injected during the execution of a single test sequence. All in all, we injected 10 different types of faults: SPADA-F1 to

Table 1. Statistical data concerning the number of EP states, EP transitions, as well as the coverage of EP states and EP transitions. All values are averages over the 700 generated test sequences per injected fault type. Values in parentheses denote standard deviations.

Injected fault	PSOPP						SPADA			
	F1	F2	F3	F4	F5	F5d	F1	F2	F3	F4
#EP States	11.68 (5.39)	15.77 (6.49)	12.30 (5.96)	13.74 (5.75)	15.41 (6.16)	14.25 (6.05)	16.42 (5.71)	14.72 (6.42)	16.62 (6.19)	15.44 (6.31)
#EP Transitions	127.42 (99.20)	222.06 (148.59)	143.31 (124.31)	169.78 (128.68)	210.22 (142.27)	183.28 (130.00)	230.83 (134.75)	197.09 (134.29)	240.56 (141.24)	212.72 (136.51)
%EP State Coverage	99.51 (2.33)	99.57 (2.08)	99.45 (2.50)	99.42 (2.71)	99.47 (2.35)	99.54 (2.31)	99.66 (1.77)	99.59 (2.15)	99.54 (2.35)	99.60 (1.98)
%EP Transition Coverage	52.31 (15.80)	45.37 (12.63)	50.57 (14.32)	47.53 (11.97)	44.76 (11.47)	47.57 (14.11)	43.98 (11.34)	47.46 (15.11)	43.52 (11.60)	45.34 (13.23)

SPADA-F4, PSOPP-F1 to PSOPP-F5, and an additional variant of PSOPP-F5, called PSOPP-F5d, which we will explain in more detail in the course of Sect. 6.3. For each type, we generated 70 test suites, each containing 10 test sequences, resulting in 700 test sequences per fault type and a total number of executed test sequences of 7000. Overall, we generated 3,679,326 test cases, corresponding to an average of 367,932.60 per fault type. As shown in Table 1, this high number of test cases allowed us to obtain an EP state coverage of more than 99% and an EP transition coverage ranging between approximately 44% and 52% on average for all fault types. To illustrate the advantages of the gray-box view, we did not abort the execution of a test sequence until the oracle found a failure using the black-box view. In the course of the execution of a single test sequence, IsoTeSO could therefore register multiple failures using the gray-box interface. On the other hand, it is not possible to detect more than one failure per test sequence using the black-box view.

6.3 Evaluation Results

As Table 2 shows, IsoTeSO was able to detect every kind of injected fault and our evaluation results support our claims made in Sect. 6.1: (1) All failures detected using the black-box view are also detected using the gray-box view, and (2) the SPADA faults cannot be detected using the black-box view. The fact that not all PSOPP faults that were disclosed using the gray-box view were also registered using the black-box view (between 0.00% and 34.44% on average) also demonstrates the problem of error masking (C-ErrorMask) in SOAS. Except for PSOPP-F4, the percentage of detected failures using the gray-box view (between 50.00% and 82.31% on average) outmatches the percentage of detected failures using the black-box view in all cases (between 31.25% and 80.00% on average). These observations clearly indicate the need for gray-box interfaces in the context of testing SO algorithms.

The relatively high number of test sequences in which no failure was detected (between 70.22% and 97.86% on average) highlights the need for directed testing that is able to deal with the huge search space in a more efficient way

Table 2. Statistical data concerning the number of agents, the number of agent groups, the number of test cases, the number of reorganizations, as well as the occurrence, detection, and depth of failures. All undetected failures (see “%Undetected failures per test sequence”) can be attributed to error masking. “#Failures” and “#Failures per test sequence” refer to the number of faulty intermediate states the corresponding SO algorithm entered (note that this information is provided by our fault injection mechanism and not by the oracle that can only check for the validity of final states, i.e., reorganization results). All values are averages over the 700 generated test sequences per injected fault type. Values in parentheses denote standard deviations.

Injected fault	PSOPP						SPADA			
	F1	F2	F3	F4	F5	F5d	F1	F2	F3	F4
#Agents	523.53 (310.53)	529.66 (292.23)	473.29 (269.35)	511.83 (287.62)	491.26 (300.28)	477.64 (294.48)	520.29 (286.29)	509.63 (289.01)	494.90 (297.69)	447.40 (279.09)
#Agent groups	39.80 (67.18)	37.42 (48.83)	20.09 (27.72)	36.05 (64.05)	30.2 (47.79)	32.74 (48.77)	31.86 (57.70)	35.12 (47.42)	31.30 (58.27)	38.85 (45.35)
%Test sequences without failure	79.97 (40.05)	84.55 (36.17)	91.42 (28.03)	96.28 (18.94)	97.86 (14.49)	70.22 (45.76)	97.13 (16.69)	84.22 (36.48)	85.69 (35.04)	90.13 (29.85)
#Test cases per test sequence	520.01 (281.50)	523.71 (275.73)	531.53 (273.61)	521.19 (278.14)	517.23 (272.25)	518.54 (270.26)	548.38 (278.18)	535.62 (271.43)	511.46 (277.29)	518.51 (282.27)
%Applied test cases per test sequence	81.37 (38.39)	85.69 (34.44)	92.74 (25.62)	96.77 (17.11)	98.62 (11.16)	73.90 (42.75)	100.00 (0.00)	100.00 (0.00)	100.00 (0.00)	100.00 (0.00)
#Reorganizations per test sequence	73.29 (64.40)	77.68 (62.08)	91.87 (59.90)	91.87 (60.24)	90.01 (56.36)	53.35 (61.19)	100.00 (58.20)	87.64 (62.46)	83.60 (60.35)	87.75 (61.30)
#Failures per test sequence	0.36 (1.89)	0.19 (0.53)	0.13 (0.52)	2.29 (16.03)	0.02 (0.15)	0.54 (1.03)	0.03 (0.17)	0.16 (0.36)	83.49 (60.39)	0.12 (0.35)
#Failures	252	130	91	1603	16	376	20	110	58443	84
%Undetected failures	44.44	17.69	34.07	97.75	50.00	40.96	0.00	0.00	99.84	9.52
%Detected failures (gray box)	55.56	82.31	65.93	2.25	50.00	59.04	100.00	100.00	0.16	90.48
%Detected failures (black box)	53.17	80.00	57.14	2.25	31.25	51.33	0.00	0.00	0.00	0.00
%Test sequences with failures detected in gray box only	4.29 (20.33)	3.70 (18.97)	13.33 (34.28)	0.00 (0.00)	34.44 (45.63)	9.74 (27.63)	100.00 (0.00)	100.00 (0.00)	100.00 (0.00)	100.00 (0.00)
%Test sequences with failures detected in black box only	0.00 (0.00)	0.93 (9.62)	0.00 (0.00)							
Depth of first detected failure	8.56 (22.82)	17.50 (62.33)	7.55 (9.12)	44.04 (109.78)	30.87 (74.51)	24.28 (75.26)	3.85 (0.49)	4.66 (1.64)	9.96 (14.38)	13.20 (16.06)
Depth of first detected failure (gray box)	8.56 (22.82)	17.63 (62.61)	7.55 (9.12)	44.04 (109.78)	30.87 (74.51)	24.28 (75.26)	3.85 (0.49)	4.66 (1.64)	9.96 (14.38)	13.20 (16.06)
Depth of first detected failure (black box)	8.77 (23.30)	13.64 (46.08)	6.08 (5.12)	44.04 (109.78)	90.27 (173.32)	22.83 (71.90)	N/A	N/A	N/A	N/A

(C-BranchingStateSpace). For PSOPP, the different numbers of applied test cases per test sequence reflect the difficulty of disclosing a specific type of fault using the black-box view. In case of SPADA, all test cases were applied since the injected faults cannot be detected using the black-box view. Another indicator for the difficulty of finding a specific fault type is the depth of the first

detected failure (i.e., the index of the test case in which the first failure was detected). Here, we see significant differences among the different SPADA and PSOPP fault types. This is because the system not only has to be pushed into a faulty intermediate state but the reorganization also has to end in a faulty state that can be detected by the oracle. This challenge becomes clearer when taking a look at the percentage of undetected failures due to error masking, which ranges between 17.69% and 97.75% for PSOPP and between 0.00% and 99.84% for SPADA. Together with the average number of reorganizations per test sequence—that, except for SPADA-F3, significantly outmatches the average number of failures per test sequence—these observations illustrate the difficulty of testing SOAS.

We observed that especially PSOPP-F5 had a relatively high number of executed test sequences without detected failures (97.86% compared to an average of 88.67% over all other types of fault). We therefore decided to use PSOPP-F5 to investigate the influence of directed testing on IsoTeSO’s ability to disclose a fault. To study this effect, we introduced an additional fault type PSOPP-F5d that is equivalent to PSOPP-F5 but uses a specific system configuration: To increase the chance of applying PSOPP’s approach exchange operator (an operation that is usually only applied in rare cases), we set the allowed number of partitions in PSOPP-F5d to $n_{min} = n_{max}$. Using this parametrization, PSOPP has no choice but to apply the random exchange or approach exchange operator, because the split/join operators increase/decrease the number of partitions by one. Note that this measure does not increase the code coverage (PSOPP also applied the approach exchange operator in case of PSOPP-F5), but the chance that the fault can be disclosed given the algorithm’s non-deterministic behavior (C-BranchingStateSpace): In approximately the same number of executed test cases, PSOPP-F5d entered a faulty intermediate state about 23 times more often than PSOPP-F5. The benefit of directed testing is further reflected in an increase of the percentage of a faulty end state in case of a faulty intermediate state from 50.00% for PSOPP-F5 to 59.04% for PSOPP-F5d. Consequently, the number of applied test sequences without detected failures dropped from 97.86% to 70.22%.

Summarizing, although IsoTeSO was able to find every type of injected fault, the high number of needed test sequences for failure detection demonstrates that the automatic generation of test suites on the basis of EPs and influence functions is especially useful for testing SOAS. This is mainly because of the non-deterministic behavior of SO algorithms (C-BranchingStateSpace). Our evaluation results also show that a combination of model-based and random generation techniques is essential to efficiently and effectively detect specific types of fault. In particular, the gray-box interface is an important feature to mitigate the effect of error masking (C-ErrorMask). However, we expect an additional white-box view to certainly further increase the potential of finding faults in SO algorithms.

7 Related Work

The necessity of testing adaptive systems has been recognized both in the testing community [30, 32, 49, 52] and in the community of adaptive systems [11, 20, 26, 35]. Run-time as well as design-time approaches have identified non-determinism and the emergent behavior as main challenges for testing adaptive systems.

Run-time approaches for testing take up the paradigm of run-time verification [17, 18, 27]. They shift testing into run-time to be able to observe and test, e.g., the adaptation to new situations. Camara and de Lemos [10] are using these concepts to consider fully integrated systems. Their testing approach focuses mainly on testing non-functional properties of the system or more precisely, the resilience of the adaptive system. The authors therefore investigate the system's adaptive capabilities by collecting and analyzing data in a simulated environment. The gained information is used as feedback for the running system. A similar approach is taken by Ramirez et al. [38], also focusing on non-functional requirements. The authors use the sampled data from a simulation to calculate a distance to expected values derived from the goal specification of the system. This information is used to adapt the system or its requirements proactively during run-time. Run-time approaches, however, are limited to tests of the fully integrated system and therefore are faced with problems like error masking which is very likely in such self-healing systems. In our layered testing approach—where IsoTeSO is placed on the interaction layer—we benefit from the piecemeal integration of the system for testing. Thus, it is possible to avoid error masking within IsoTeSO by testing the SO algorithms isolated.

An important difference to the mentioned work is that we use these techniques for finding failures instead of analyzing the current system state for generating feedback for the adaptation. Still, we also use the basic concepts of run-time testing. The CEI allows us to split the evaluation into the three responsibilities R-Detect, R-Solution, and R-Distribution which in turn enable us to evaluate the runs without the evaluation of complex system states on the system level. As our evaluation shows, the CEI-based testing approach is especially beneficial in the context of self-organization.

Design-time approaches like [28, 30, 32, 45, 52] test the systems in a classical manner during the development. All these approaches are considering some dedicated parts of the system. Consequently, it is not possible to give evidence about the correct functionality of the overall system. Zhang et al. [52] compose their tests towards a fully integrated system test, but they do not consider adaptivity or SO explicitly since they focus on testing the correct execution of plans within multi-agent systems. Nguyen [30] promote an approach for a component test suite (where the components correspond to agents), but do not consider interaction or organization between the agents as it would be necessary for SO.

The evaluation of the test results, i.e., the application of a test oracle for adaptive behavior is only considered by Fredericks et al. [19, 20] and Nguyen et al. [31]. Both approaches are relying on goals reflecting the requirements of

the system that are somewhat loosened in order to reflect the ever-changing environment the agents have to adapt to: The approaches mitigate the goals with the *RELAXed* approach [48] or consider soft goals that do not need to hold at all time. Consequently, the decision of the test oracle is rather fuzzy. In our approach the definition of correct and in-correct behavior is given by the CCB that enables us to decide whether a failure occurs or not.

Probabilistic Test Models of Environmental Changes and Influences. There are several approaches to tackle uncertainties about the expected usage of a SuT within testing models. They can be summed up under the idea of *operational profiles* [43]. The information within these test models represents the user's behavior in a probabilistic model. For this purpose, different techniques for generating and using these profiles have been provided.

Sammodi et al. [40], e.g., generate usage profiles, as they call them, by monitoring the user's interaction with the system and deriving the profiles for the observed usage afterward. One of our possibilities to establish EPs also follows this monitoring and analysis process with the difference that we are not monitoring users of the system. Instead we monitor the whole system environment which includes all influences on the SOuT. Another approach to design models of the usage is presented by Samih et al. [39]. They enrich the models by introducing capabilities in order to model variants of specific features, i.e., product features, to form test models for product line engineering. The approach made by Ehlers et al. [16] focuses on using the usage profiles for detecting anomalies in adaptive systems in order to use the information about the anomalies during the adaptation process. Besides focusing on handling user behavior, there is also some work on representing the behavior of other system components in the test model, like Popovic and Kovacevic [34]. The authors use the models for protocol testing and therefore represent valid and invalid communication between components.

Operational profiles thus are an established technique for modeling uncertain behavior—mainly of the user—for designing test models and for evaluation purposes. In IsoTeSO, we use environment profiles which are based on a similar concept to deal with the complexity of the ever-changing environment of SOAS by reducing its state space using a probabilistic approach.

Isolated Testing of Component-Based Systems. Our presented concepts for a framework for isolated testing of SO algorithms are related to methods and techniques in the research area of isolated testing of component-based systems. A recent approach in this area by Thillen et al. [46] promotes the “tester in the middle” idea that aims at improving testing of distributed components depending on other components in the system. Their application area is the testing of network components. For this purpose, they model dependencies within the network to be able to build mock-ups out of this models. Bauer and Eschbach [8] propose a statistical strategy for isolated testing of component-based systems.

Their approach is based on state-based models that are used to generate interaction test models. The goal is to test the interactions and functionalities within the system under test which is composed of several system components. Yao and Wang [51] present a framework for testing distributed software components that provides an environment to allow a client-side software component to define tests for a black-box component published on the server-side. The framework focuses on automatic test execution without considering explicitly the generation and evaluation of the tests. Wu et al. [50] propose JATA, a testing language for distributed components enabling the use JUNIT in the context of service-oriented systems and offering support for a message-oriented middleware; like Yao and Wang [51] it focuses on the execution of component tests on web services. In contrast to the approaches in [50, 51], IsoTeSO offers a complete approach from test generation over execution to test evaluation with a focus on agent-based systems and functional testing of single SO algorithms.

Our approach for isolated testing of SO algorithms is an efficient combination of model-based techniques using the concepts of isolated testing and probabilistic modeling in order to tackle the challenges of testing SOAS. To our knowledge, there is no approach extending both of these techniques to SO algorithms.

8 Conclusion and Future Work

We motivated the necessity of testing SOAS and outlined an approach that copes with the complexity arising from the characteristics of these systems. We introduced the framework IsoTeSO for testing SO algorithms that is an important element coping up with this challenge. To be able to test SO algorithms in a systematic and automatic fashion, we identified four challenges that are addressed: error masking among the algorithms of SOAS, interleaved feedback loops which distort test results, the oracle problem in the context of SO, and the ramified state space spanned by SO algorithms. We were able to show that it is possible to meet these challenges by isolating the SO algorithms and applying a model-based testing approach. The resulting framework encompasses automatic test suite generation, execution, and evaluation within a controlled environment. Environment profiles are used to abstract from concrete environments and allow for generating large test sequences. This technique showed to be extremely valuable since a lot of test runs are necessary to find a trace through a SOAS that actually reveals a failure.

We evaluated IsoTeSO on the basis of two different partitioning-based SO algorithms in an existing smart-grid application. Our results demonstrate that an efficient combination of model-based and random test case generation techniques allows to find different kinds of failures in an acceptable time limit. In particular, the gray-box view turned out to be very important to reduce the effect of error masking.

Future work focuses on three aspects: first, *extending the evaluation* of the test run results enabling *fault diagnosis*; second, gaining more *control on the execution* to increase *reproducibility* of the test results; and third, using IsoTeSO to

investigate non-functional aspects of the system, i.e., to evaluate the performance of SO-algorithms.

For the extension toward fault diagnosis, we aim at combining the different test sequences of a specific test suite into a *test sequence evaluation tree*. For this purpose, we will try to identify equal prefixes of the test sequences. If two or more test sequences share a common prefix, they will share a common path in the tree. Thus, it is possible to unfold overlaps between executed and evaluated test sequences and map them into a tree model. This tree could be used for fault localization, optimization of later test generations, or visualization of the results for a deeper understanding.

To increase the *reproducibility* of the test results, the test system has to ensure the following properties: (1) a random seed has to be used in the whole test system; (2) the scheduling of the underlying execution platform has to be controlled in a reproducible way; and (3) the concurrent execution of a distributed test system has to be synchronized in a deterministic way. There are different techniques to address (1) to (3) in order to detect so-called “Mandelbugs”⁷ (cf. the CHESS tool by Musuvathi et al. [29] or the empirical study on this topic by Thomson et al. [47]). These could be used to provide an adequate underlying platform for IsoTeSO. Although designing such a platform was not in the scope of this paper, we met the points (1) and partially (3). As we perform our tests within the limits of the underlying execution platform (Java), which does not affect the ability to detect faults but might affect the reproducibility (at least in some cases), we currently do not meet (2) and achieve (3) only on the basis of a stepwise execution model. We are going to extend our experiments with a continuous execution model to evaluate differences in the test results and its effects on the reproducibility as well as the ability of IsoTeSO to execute tests in this environment.

Besides investigating functional tests we already started to use IsoTeSO for the evaluation of performance criteria. In [13], we showed its ability to perform evaluations on non-functional aspects of SO algorithms and developed a set of five major requirements for metrics that evaluate the performance of SO algorithm within IsoTeSO. The development of the metrics and the integration into the framework is a further topic of future work in testing self-organizing, adaptive systems.

Acknowledgment. This research is sponsored by the research project *Testing Self-Organizing, adaptive Systems (TeSOS)* of the German Research Foundation.

⁷ According to Grottke and Trivedi [21] a Mandelbug is “[a] fault whose activation and/or error propagation are complex, where ‘complexity’ can take [the following form]: [...] The activation and/or error propagation depend on interactions between conditions occurring inside the application and conditions that accrue within the system-internal environment of the application [...]”.

References

1. Anders, G., Siefert, F., Mair, M., Reif, W.: Proactive guidance for dynamic and cooperative resource allocation under uncertainties. In: Proceedings of the 8th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO). IEEE Computer Society (2014)
2. Anders, G., Siefert, F., Msadek, N., Kiehaber, R., Kosak, O., Reif, W., Ungerer, T.: TEMAS - a trust-enabling multi-agent system for open environments. Technical report 2013–04, Universität Augsburg (2013). <http://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/docId/2311>
3. Anders, G., Siefert, F., Reif, W.: A particle swarm optimizer for solving the set partitioning problem in the presence of partitioning constraints. In: Proceedings of the 7th International Conference on Agents and Artificial Intelligence (ICAART). SciTePress (2015)
4. Anders, G., Siefert, F., Steghöfer, J.-P., Reif, W.: A decentralized multi-agent algorithm for the set partitioning problem. In: Rahwan, I., Wobcke, W., Sen, S., Sugawara, T. (eds.) PRIMA 2012. LNCS (LNAI), vol. 7455, pp. 107–121. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32729-2_8
5. Anders, G., Steghöfer, J.P., Klejnowski, L., Wissner, M., Hammer, S., Siefert, F., Seebach, H., Bernard, Y., Reif, W., Müller-Schloer, C., André, E.: Reference architectures for trustworthy energy management, desktop grid computing applications, and ubiquitous display environments. Technical report 2013–05, Universität Augsburg (2013). <http://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/docId/2303>
6. Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M., Pasareanu, C., Roşu, G., Sen, K., Visser, W., et al.: Combining test case generation and runtime verification. *Theoret. Comput. Sci.* **336**(2), 209–234 (2005)
7. Balas, E., Padberg, M.W.: Set partitioning: a survey. *SIAM Rev.* **18**(4), 710–760 (1976)
8. Bauer, T., Eschbach, R.: Enabling statistical testing for component-based systems. In: Fähnrich, K.P., Franczyk, B. (eds.) GI Jahrestagung. LNI, vol. 176, pp. 357–362. GI (2010)
9. Binder, R.V.: *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley, Boston (1999)
10. Cámarra, J., de Lemos, R.: Evaluation of resilience in self-adaptive systems using probabilistic model-checking. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 53–62 (2012)
11. Cheng, B.H.C., et al.: Software engineering for self-adaptive systems: a research roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) *Software Engineering for Self-Adaptive Systems*. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_1
12. Eberhardinger, B., Anders, G., Seebach, H., Siefert, F., Reif, W.: A framework for testing self-organisation algorithms. In: 37 Treffen der GI Fachgruppe TAV, vol. 35:1. Softwaretechnik-Trends der Gesellschaft für Informatik (2015)
13. Eberhardinger, B., Anders, G., Seebach, H., Siefert, F., Reif, W.: A research overview and evaluation of performance metrics for self-organization algorithms. In: Proceedings of the 9th IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshop (SASOW), pp. 122–127. IEEE Computer Society (2015)

14. Eberhardinger, B., Seebach, H., Knapp, A., Reif, W.: Towards testing self-organizing, adaptive systems. In: Merayo, M.G., de Oca, E.M. (eds.) ICTSS 2014. LNCS, vol. 8763, pp. 180–185. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44857-1_13
15. Eberhardinger, B., Steghöfer, J.P., Nafz, F., Reif, W.: Model-driven synthesis of monitoring infrastructure for reliable adaptive multi-agent systems. In: Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering (ISSRE), pp. 21–30. IEEE Computer Society (2013)
16. Ehlers, J., van Hoorn, A., Waller, J., Hasselbring, W.: Self-adaptive software system monitoring for performance anomaly localization. In: Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC), pp. 197–200. ACM (2011)
17. Falcone, Y., Jaber, M., Nguyen, T.-H., Bozga, M., Bensalem, S.: Runtime verification of component-based systems. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 204–220. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24690-6_15
18. Filieri, A., Ghezzi, C., Tamburrelli, G.: A formal approach to adaptive software: continuous assurance of non-functional requirements. Formal Asp. Comp. **24**(2), 163–186 (2012)
19. Fredericks, E.M., DeVries, B., Cheng, B.H.C.: Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty. In: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 17–26. ACM (2014)
20. Fredericks, E.M., Ramirez, A.J., Cheng, B.H.C.: Towards run-time testing of dynamic adaptive systems. In: Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 169–174. IEEE (2013)
21. Grottke, M., Trivedi, K.S.: A classification of software faults. J. Reliab. Eng. Assoc. Jpn. **27**(7), 425–438 (2005)
22. Güdemann, M., Nafz, F., Ortmeier, F., Seebach, H., Reif, W.: A specification and construction paradigm for organic computing systems. In: Brueckner, S.A., Robertson, P., Bellur, U. (eds.) Proceedings of the 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems, pp. 233–242. IEEE Computer Society (2008)
23. Hierons, R.M.: Oracle for distributed testing. IEEE Trans. Softw. Eng. **38**(3), 629–641 (2012)
24. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Proceedings of the IEEE International Conference on Neural Networks, vol. 4, pp. 1942–1948 (1995)
25. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
26. de Lemos, R., et al.: Software engineering for self-adaptive systems: a second research roadmap. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 1–32. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_1
27. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebraic Program. **78**(5), 293–303 (2009). Proceedings of the 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS)
28. Luckey, M., Thanos, C., Gerth, C., Engels, G.: Multi-staged quality assurance for self-adaptive systems. In: Proceedings of the 6th International Conference on Self-Adaptive and Self-Organizing Systems Workshop (SASOW), pp. 111–118 (2012)

29. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing Heisenbugs in concurrent programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI), pp. 267–280. USENIX Association (2008)
30. Nguyen, C.D.: Testing techniques for software agents. Ph.D. thesis, Università di Trento (2009)
31. Nguyen, C.D., Marchetto, A., Tonella, P.: Automated oracles: an empirical study on cost and effectiveness. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Proceedings of the Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE), pp. 136–146. ACM (2013)
32. Padgham, L., Thangarajah, J., Zhang, Z., Miller, T.: Model-based test oracle generation for automated unit testing of agent systems. *IEEE Trans. Softw. Eng.* **39**(9), 1230–1244 (2013)
33. Pezzé, M., Young, M.: Software Testing and Analysis: Process, Principles and Techniques. Wiley, New York (2005)
34. Popovic, M., Kovacevic, J.: A statistical approach to model-based robustness testing. In: Proceedings of the 14th IEEE Conference and Workshops on Engineering of Computer-Based Systems (ECBS), pp. 485–494 (2007)
35. Püschel, G., Götz, S., Wilke, C., Aßmann, U.: Towards systematic model-based testing of self-adaptive software. In: Proceedings of the 5th International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE), pp. 65–70 (2013)
36. Püschel, G., Götz, S., Wilke, C., Piechnick, C., Aßmann, U.: Testing self-adaptive software: requirement analysis and solution scheme. *Int. J. Adv. Softw.* **7**(1 & 2), 88–100 (2014)
37. Ramchurn, S.D., Vytelingum, P., Rogers, A., Jennings, N.R.: Putting the “smarts” into the smart grid: a grand challenge for artificial intelligence. *Commun. ACM* **55**(4), 86–97 (2012)
38. Ramirez, A.J., Jensen, A.C., Cheng, B.H.C., Knoester, D.B.: Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems. In: Alexander, P., et al. (eds.) Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 568–571. IEEE (2011)
39. Samih, H., Le Guen, H., Bogusch, R., Acher, M., Baudry, B.: An approach to derive usage models variants for model-based testing. In: Merayo, M.G., de Oca, E.M. (eds.) ICTSS 2014. LNCS, vol. 8763, pp. 80–96. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44857-1_6
40. Sammodi, O., Metzger, A., Franch, X., Oriol, M., Marco, J., Pohl, K.: Usage-based online testing for proactive adaptation of service-based applications. In: Proceedings of the 35th IEEE Computer Software and Applications Conference (COMPSAC), pp. 582–587 (2011)
41. Schmeck, H., Müller-Schloer, C., Çakar, E., Mnif, M., Richter, U.: Adaptivity and self-organization in organic computing systems. *ACM Trans. Auton. Adapt. Syst.* **5**(3), 10 (2010)
42. Scott, P., Thiébaux, S., van den Briel, M., Van Hentenryck, P.: Residential demand response under uncertainty. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 645–660. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40627-0_48
43. Smidts, C., Mutha, C., Rodríguez, M., Gerber, M.J.: Software testing with an operational profile: OP definition. *ACM Comput. Surv.* **46**(3), 39:1–39:39 (2014)

44. Steghöfer, J.P., Anders, G., Siefert, F., Reif, W.: A system of systems approach to the evolutionary transformation of power management systems. In: Proceedings of Informatik 2013 - Workshop on “Smart Grids”. Lecture Notes in Informatics. Bonner Kölle Verlag (2013)
45. Stott, D.T., Flooring, B., Burke, D., Kalbarczyk, Z., Iyer, R.K.: NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors. In: Proceedings of the IEEE International Computer Performance and Dependability Symposium (IPDS), pp. 91–100. IEEE (2000)
46. Thillen, F., Mordinyi, R., Biffl, S.: Isolated testing of software components in distributed software systems. In: Winkler, D., Biffl, S., Bergmann, J. (eds.) SWQD 2014. LNBI, vol. 166, pp. 170–184. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-03602-1_11
47. Thomson, P., Donaldson, A.F., Betts, A.: Concurrency testing using schedule bounding: an empirical study. SIGPLAN Not. **49**(8), 15–28 (2014)
48. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.: RELAX: incorporating uncertainty into the specification of self-adaptive systems. In: Proceedings of the 17th IEEE International Requirements Engineering Conference (RE), pp. 79–88. IEEE Computer Society (2009)
49. Wotawa, F.: Adaptive autonomous systems – from the system’s architecture to testing. In: Hähnle, R., Knoop, J., Margaria, T., Schreiner, D., Steffen, B. (eds.) ISoLA 2011. CCIS, pp. 76–90. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34781-8_6
50. Wu, J., Yang, L., Luo, X.: Jata: a language for distributed component testing. In: 15th Asia-Pacific Software Engineering Conference (APSEC), pp. 145–152 (2008)
51. Yao, Y., Wang, Y.: A framework for testing distributed software components. In: Proceedings of the IEEE Conference Electrical and Computer Engineering, pp. 1566–1569. IEEE (2005)
52. Zhang, Z., Thangarajah, J., Padgham, L.: Model based testing for agent systems. In: Decker, et al. (eds.) Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 1333–1334. IFAAMAS (2009)

Using Runtime Quantitative Verification to Provide Assurance Evidence for Self-Adaptive Software

Advances, Applications and Research Challenges

Radu Calinescu^{1(✉)}, Simos Gerasimou¹, Kenneth Johnson²,
and Colin Paterson¹

¹ Department of Computer Science, University of York, York, UK
Radu.Calinescu@york.ac.uk

² School of Computer and Mathematical Science,
Auckland University of Technology, Auckland, New Zealand

Abstract. Providing assurance that self-adaptive software meets its dependability, performance and other *quality-of-service* (QoS) requirements is a great challenge. Recent approaches to addressing it use formal methods at runtime, to drive the reconfiguration of self-adaptive software in provably correct ways. One approach that shows promise is *runtime quantitative verification* (RQV), which uses quantitative model checking to reverify the QoS properties of self-adaptive software after environmental, requirement and system changes. This reverification identifies QoS requirement violations and supports the dynamic reconfiguration of the software for recovery from such violations. More importantly, it provides irrefutable *assurance evidence* that adaptation decisions are correct. In this paper, we survey recent advances in the development of efficient RQV techniques, the application of these techniques within multiple domains and the remaining research challenges.

1 Introduction

Self-adaptive software aims to operate correctly in the presence of uncertainty arising from simplifying assumptions, unknown parameters, goal changes, component failures and many other causes. The demand for this capability is growing fast. In domains ranging from healthcare and transportation to finance and defence, the ability to cope with uncertainty could lead to a more effective use of new technologies. Through leveraging technologies such as the Internet-of-Things, unmanned vehicles and cloud services, self-adaptive software could enable the development of new applications of significant economic and societal benefit. Some examples of these include emergency management (e.g., natural-disaster response), environmental surveillance (e.g., monitoring of storms and pollution levels), and telehealth and telecare. Given the safety-critical nature of such applications, achieving this vision requires assurances that self-adaptive software meets its functional and quality-of-service (QoS) requirements.

Traditionally, assurances for software used in safety-critical applications are based on evidence obtained through verification and validation activities carried out during the design, implementation and testing stages of the software lifecycle. A set of mathematically based techniques called *formal methods* are particularly useful for this purpose [72]. Another effective method to obtain assurance evidence is testing [25]. Unfortunately, the uncertainty that self-adaptive software has to cope with in its production environment means that neither approach can provide complete assurance evidence for such software before deployment.

To address this limitation, recent research has advocated the use of formal methods to drive the reconfiguration of self-adaptive software [16] or changes to its logic [28, 43] in provably correct ways. A promising approach from the former area is *runtime quantitative verification* (RQV) [10]. Quantitative verification [55] is a formal technique for establishing the reliability, performance and other QoS properties of stochastic systems at design time, and RQV represents its runtime counterpart. To this end, RQV uses continually updated stochastic models of a self-adaptive system to reverify its compliance with QoS requirements after environmental and system changes. This reverification identifies or predicts QoS requirement violations, and supports the self-adaptation of the software for recovery from, or prevention of, such violations. In doing so, it provides evidence that QoS requirements are met and that adaptation decisions are correct.

In this paper we survey recent advances in the development of efficient RQV techniques, the application of these techniques within multiple domains, and the remaining research challenges. We start with an overview of runtime quantitative verification and its use in developing self-adaptive software in Sect. 2. In Sect. 3 we summarise key developments in making RQV applicable in a broader range of domains and scenarios. Two areas of such developments are covered. The first area, presented in Sect. 3.1, comprises results on improving the efficiency of RQV. These results overcome some of the overheads traditionally associated with formal verification. The second area, described in Sect. 3.2, includes recent work on improving the accuracy of the models that RQV operates with. In Sect. 4, we discuss applications of RQV from domains including service-based systems, management of cloud computing infrastructure, unmanned vehicles, and dynamic power management. We conclude the paper with a brief summary and a discussion of remaining research challenges in Sect. 5.

2 Overview

2.1 Quantitative Verification

Quantitative verification [4, 55, 58] is a mathematically based technique for analysing the correctness, reliability, performance and other QoS properties of systems characterised by stochastic behaviour. The technique carries out analysis on finite state-transition models comprising states that correspond to different system configurations and edges associated with the transitions that are possible between these states. Depending on the analysed QoS properties, the edges

are annotated with transition probabilities or transition rates; additionally, the model states and transitions may be labelled with costs/rewards. The types of models with probability-annotated transitions include discrete-time Markov chains (DTMCs) and Markov decision processes (MDPs), while the edges of continuous-time Markov chains (CTMCs) are annotated with transition rates.

Given one of these models and a QoS property specified formally in a variant of temporal logic extended with probabilities and costs/rewards, the technique analyses the model exhaustively to evaluate the property. Examples of properties that can be established using the technique include the probability that a software component operates correctly, the expected reliability of a composite service, and the expected energy consumed by an embedded system. Quantitative verification is typically performed entirely automatically by tools termed *probabilistic model checkers*. Widely used probabilistic model checkers include PRISM [56], MRMC [52] and Ymer [73]. These tools use model checking algorithms that are provably correct. Therefore, if the implementation of these algorithms is error free and the verified model is an accurate representation of the analysed system, QoS properties established by a probabilistic model checker can be used as assurance evidence.

Example 1. Consider a unmanned underwater vehicle (UUV) equipped with $n \geq 1$ sensors that can measure the same attribute of the marine environment (e.g., current, salinity or thermocline). Suppose that the i -th sensor takes measurements with a variable rate r_i and consumes energy e_i for each measurement. Every measurement is followed by operations that prepare the sensor for the next measurement, and these operations are carried out with a rate r_i^{prep} . The probability p_i that a measurement is accurate depends on the configurable UUV speed $sp \in [0, 5\text{m/s}]$: $p_i = 1 - \alpha_i sp$, where $\alpha_i \in (0, 0.15)$ is a sensor-specific accuracy factor. A configurable parameter $x_i \in \{0, 1\}$ can be used to switch the sensor on and off in order to save energy ($x_i = 1$ when the sensor is operational and $x_i = 0$ when the sensor is switched off). However, switching the sensor on or off consumes an amount of energy e_i^{on} or e_i^{off} , respectively.

Figure 1a shows the CTMC model of sensor i -th, adapted from [40]. From the initial state s_0 , the model transitions to state s_1 if the sensor is switched on, or to state s_4 if the sensor is switched off. The rates for these transitions are r_i^{on} and r_i^{off} , respectively. The CTMC models a session during which the sensor is either operational or switched off, so a switched-off sensor remains in state s_4 indefinitely. In contrast, an operational sensor takes measurements with rate r_i and therefore leaves state s_1 with this rate. The next state is either s_2 , if the measurement was accurate (with probability p_i), or s_3 otherwise. Irrespective of whether the measurement was successful or not, the model transitions back to state s_1 with the rate r_i^{prep} associated with the operations that prepare the next measurement.

The CTMC transitions associated with sensor operations that consume energy are annotated with the energy used by these operations (shown in red/non-shaded squares in Fig. 1a). Similarly, the transitions that corresponds to

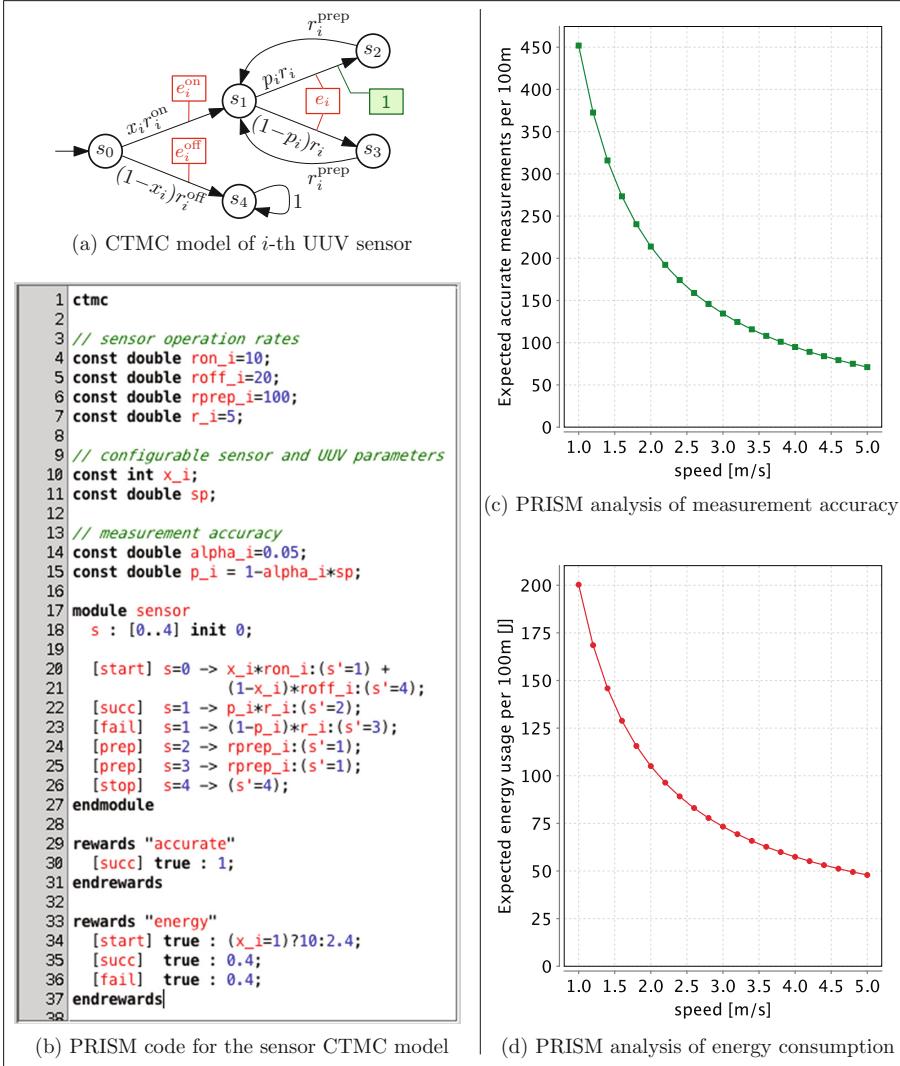


Fig. 1. Modelling and quantitative verification of UUV sensor properties (Color figure online)

a successful (i.e., accurate) measurement is annotated with a *reward* of 1 (shown as a green/shaded square in Fig. 1a).

Figure 1b illustrates the representation of the sensor CTMC model in the high-level modelling language of the probabilistic model checker PRISM [56]. The model states and transitions are specified within **module ... endmodule** PRISM constructs, and the cost/reward annotations for the CTMC model in Fig. 1a are encoded as **rewards ... endrewards** PRISM structures. The graphs in Fig. 1c, d depict the expected number of accurate measurements and the

expected energy consumption when sensor i is operational (i.e., $x_i = 1$) for each 100 m travelled by the UUV, as a function of the UUV speed. These results were established by using PRISM to analyse the two QoS properties expressed in a reward-augmented variant of continuous stochastic logic (CLS) [3]. In particular, we used the reward-augmented CSL formulae $R_{=?}^{“accurate”}[C \leq t]$ and $R_{=?}^{“energy”}[C \leq t]$, with $t = 100/sp$. These two formulae represent the expected number of accurate measurements and the expected energy, respectively, “cumulated” up to the time t within which the UUV travels 100 m at speed sp .

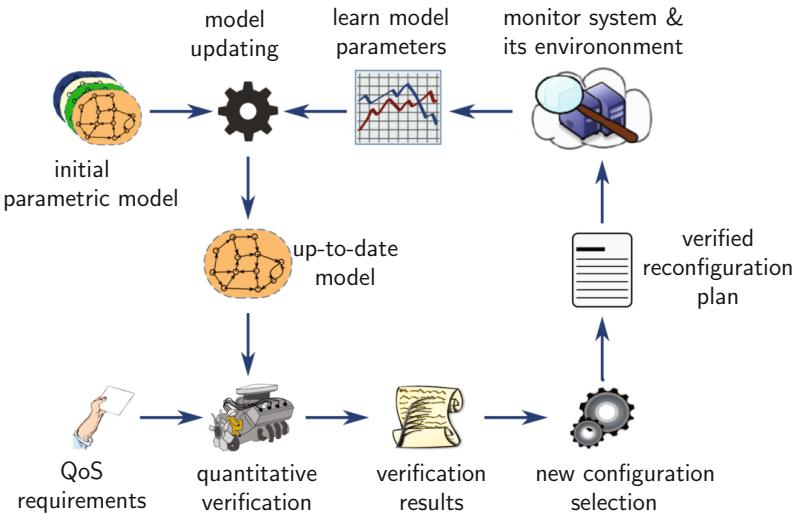


Fig. 2. Runtime quantitative verification

2.2 Runtime Quantitative Verification

Like most formal verification techniques, quantitative verification is traditionally used in off-line settings, e.g. to evaluate the performance-cost tradeoffs of alternative system designs and to establish if existing systems meet their QoS requirements. In the latter case, systems in violation of their QoS requirements undergo off-line maintenance, and are eventually replaced with suitably modified system versions. This approach does not meet the demands of emerging application scenarios in which systems need to be continually verified as they adapt autonomously to changes in their operating environment [7]. To address this need for continual verification, a runtime variant of the technique was introduced in [18, 33] and further refined by recent research [10, 35, 36].

Figure 2 illustrates the use of runtime quantitative verification for the continual verification and reconfiguration of a self-adaptive system. The system and its environment are monitored continually, and relevant changes are identified and

quantified using fast on-line learning techniques. The identification of the current environmental parameters and system state supports the selection of a suitable concrete model from a family of parametric system models associated with different “scenarios”, and typically provided by the system developers. As an example, Bayesian learning techniques are used in [13, 33] to monitor the changing probabilities of successful service invocation for a service-based system, and thus to determine the transition probabilities of a parametric Markovian chain that models the system.

The model selected through the monitoring process described above is then analysed using quantitative verification in order to identify (and in some cases to predict) violations of QoS requirements related to the system response time, availability and cost. When QoS requirement violations are identified or predicted, the verification results support the synthesis of a verified reconfiguration plan. This plan comprises adaptation steps whose execution ensures that the system will continue to satisfy its QoS requirements despite the changes identified in the monitoring step. Note that runtime changes to the QoS requirements of the system are also possible, although such changes currently need to be provided by a human operator.

The verification results and the verified reconfiguration plan produced by the RQV-driven closed control loop in Fig. 2 represent assurance evidence that the self-adaptive system will meet its QoS requirements after reconfiguration. As in the case of design-time quantitative verification, this evidence is valid as long as the following assumptions are met:

1. the quantitative verification is carried out by a probabilistic model checker that uses verification algorithms for which formal correctness proofs are available;
2. the implementation of the verification algorithms is error free;
3. the verified up-to-date model provides an accurate representation of the evolving behaviour of the self-adaptive system;
4. the QoS requirements are correctly translated into the probabilistic temporal logic used in quantitative verification.

Other approaches have also been used to model and analyse the QoS properties of self-adaptive software systems at runtime. In particular, the modelling of the QoS requirements of adaptive service-based systems as an optimisation problem that can be solved using linear or convex programming has been proposed, e.g. in [2, 23, 74]. RQV is complementary to these effective approaches, as it can be applied to self-adaptive systems whose QoS requirements can be modelled stochastically and cannot be formulated as a linear or convex programming problem.

Example 2. Suppose that the UUV system from Example 1 is required to adapt to changes in the measurement rates r_1, r_2, \dots, r_n of its n sensors and to sensor failures by dynamically adjusting

Table 1. QoS requirements for the self-adaptive UUV system

ID	Informal description	Formal specification
R1	<i>Performance</i> : “At least 300 accurate measurements must be taken for each 100 m travelled by the UUV.”	$R_{\geq 300}^{“\text{accurate}”} [C^{\leq 100/sp}]$
R2	<i>Energy use</i> : “The energy consumed by the sensors must not exceed 400 Joules per 100 m travelled by the UUV.”	$R_{\leq 400}^{“\text{energy}”} [C^{\leq 100/sp}]$
R3	<i>Utility</i> : “Subject to R1 and R2 being met, the UUV must use a configuration that maximises $\text{utility}(x_1, x_2, \dots, x_n, sp) = w_1 sp + w_2/E,$ where E is the energy used by the sensors per 100 m travelled by the UUV, and the weights $w_1, w_2 > 0$ express the desired trade-off between mission completion time and battery usage.	find argmax utility $(x_1, x_2, \dots, x_n, sp)$ such that R1 \wedge R2

- the UUV speed sp
- the set of operational sensors (i.e., the sensors whose switch on/off parameter x_1, x_2, \dots, x_n has value 1)

so that the UUV meets the QoS requirements in Table 1. The RQV closed control loop from Fig. 2 can be used to achieve this adaptation as follows. First, an initial parametric model of the n UUV sensors can be built through the parallel combination of CTMC models of the individual sensors, each of which has the structure from Fig. 1a. Next, the QoS requirements for the UUV system are formalised as shown in the last column of Table 1.

The CTMC model of the system is parameterised by the measurement rates r_1, r_2, \dots, r_n of the n sensors, so the actual values of these rates are obtained through observing the runtime behaviour of the UUV sensors. The actual measurement rates are then used to obtain an up-to-date CTMC model that reflects that actual behaviour of the sensors. This model is then used as input for the quantitative verification step. The verification results comprise the values of the QoS properties associated with system requirements and the possible configurations that can be selected for the system. For the self-adaptive UUV system, the QoS properties associated with requirements **R1–R3** from Table 1 are the number of accurate measurements and the sensor energy consumption per 100 m of distance travelled by the UUV. Figure 3a, b show the verification results for these properties for a UUV with $n = 2$ sensors with current measurement rates $r_1 = 5 \text{ s}^{-1}$ and $r_2 = 9 \text{ s}^{-1}$.

Finally, the new configuration selection stage of the RQV closed control loop selects a reconfiguration plan in two steps. In the first step, it discards the configurations that violate requirements **R1** and/or **R2**. These configurations correspond to the shaded areas from Fig. 3a, b. In the second step, the *utility* of the remaining configurations (i.e., those that satisfy requirements **R1** and **R2**) is computed as illustrated in Fig. 3c (these results were obtained using the weights $w_1 = 1$ and $w_2 = 200$ in the *utility* formula from Table 1). At the end of

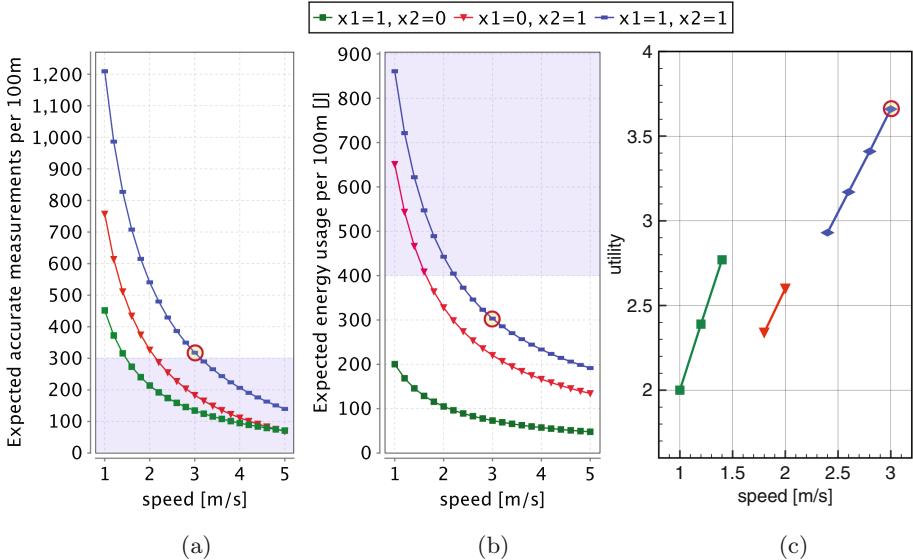


Fig. 3. Verification results for (a) requirement **R1** and (b) requirement **R2** from Table 1, and (c) *utility* of the valid configurations of a two-sensor UUV

this step, the configuration that maximises the system *utility* (shown in a small circle in Fig. 3c, and also in Fig. 3a, b), is selected and used to reconfigure the UUV system.

3 Recent Advances

This section summarises recent research on methods, frameworks and tools that improve runtime quantitative verification with a view to extending its applicability to more complex and larger self-adaptive systems. To this end, we focus on research efforts to increase the efficiency of runtime quantitative verification and the accuracy of the stochastic models it operates with.

3.1 Efficient Runtime Quantitative Verification

Since it was introduced in [18, 33], RQV has been used to develop self-adaptive systems in case studies from numerous application domains (see Sect. 4). Notwithstanding its merits, the approach is affected by *state-space explosion*, a problem common to all model checking techniques, where the size of the model increases exponentially with the size of the system. This makes the computation and memory overheads of RQV unacceptable for large and/or complex self-adaptive systems.

To address this limitation, the research community has recently proposed RQV variants that operate with reduced overheads and improved scalability.

In this section, we present the most promising approaches produced by this research. We organise the new approaches based on the procedure used to improve RQV efficiency into: (1) compositional and incremental; (2) parametric; (3) conventional software engineering; and (4) decentralised.

Compositional and Incremental RQV. Using quantitative verification to establish the QoS properties of large, complex systems at runtime is very challenging because of the size of the models that must be analysed. However, these systems typically consist of interoperating components. Accordingly, their large models are obtained through the parallel composition of smaller component models. This characteristic is exploited by *compositional verification*, a model checking technique originally proposed by Pnueli [63] in the area of model checking. In Pnueli's formulation, the technique establishes that the parallel composition of two models $M_1 \parallel M_2$ satisfies a global property \mathcal{G} by independently analysing two premises. The former premise to establish is that M_2 satisfies \mathcal{G} when the component modelled by M_2 belongs to a system that satisfies an *assumption* \mathcal{A} . The latter premise is that \mathcal{A} is satisfied by the other parts of the system (i.e., by M_1) under all circumstances. Pnueli [63] expresses this formally by generalising Hoare's triple notation [50]:

$$\frac{\langle \text{true} \rangle M_1 \langle \mathcal{A} \rangle, \langle \mathcal{A} \rangle M_2 \langle \mathcal{G} \rangle}{\langle \text{true} \rangle M_1 \parallel M_2 \langle \mathcal{G} \rangle}. \quad (1)$$

This compositional verification technique is called *assume-guarantee reasoning*.

A probabilistic variant of assume-guarantee reasoning was introduced in [57]. Probabilistic assume-guarantee operates with probabilistic automata [64], a class of nondeterministic models that generalise Markov decision processes. The technique analyses probabilistic safety properties of the form $\langle \mathcal{X} \rangle_{\geq p}$, which are satisfied by a model M iff the minimum probability that \mathcal{X} is satisfied over all possible ways to resolve the nondeterminism in M is at least p . Extensive experiments reported in [57] show that the time required to verify safety-related QoS properties of large models using probabilistic assume-guarantee are often orders of magnitude smaller than the time taken by the non-compositional verification of the same properties. This improvement in efficiency makes RQV applicable to much larger systems than previously possible.

A complementary technique called *incremental verification* is advocated in [42] for systems that require reverification after localised changes. The aim is to identify and execute a minimal sequence of reverification steps after each runtime change in a component-based system, reusing existing results to reduce the verification time whenever possible. This can be achieved by exploiting the dependencies between the system components, derived from the system architecture and expressed symbolically as a *dependency tree*. By associating the component models with probabilistic safety properties, probabilistic assume-guarantee verification steps are applied according to a depth-first traversal of the tree. The resulting *compositional verification task* is a sequence of verification steps where each step is performed using the results of some or all of the previous steps as

assumptions. The technique is extended and reified in [17] and is applied to the RQV of dynamically changing cloud computing infrastructure.

The tool-supported *Incremental Verification STrategy* (INVEST) framework [51] for the efficient incremental verification of component-based systems extends these results further. INVEST comprises three layers:

1. a generic incremental verification engine that identifies the minimal sequence of components requiring reverification after a change;
2. an assume-guarantee model checker that performs the reverification based on the sequence identified by the engine;
3. a domain-specific adaptor that connects the framework to component-based systems, for the incremental verification of their probabilistic safety properties.

The experiments reported in [51] show that INVEST can establish probabilistic safety properties in 10–60% of the time taken by compositional quantitative verification for a wide range of change scenarios.

A foundational approach to incremental verification is introduced in [59]. The approach decomposes the underlying graph of a Markov model into groups of states comprising a path between any two states, and which is maximal with respect to this property. These groups of states are called *strongly connected components* (SCCs). The approach evaluates each SCC individually using value iteration, and uses the SCC-level evaluations to derive the system-level verification results. When a change in state transition probabilities occurs, the technique reverifies only the SCCs affected by the change, and reuses the verification results corresponding to unaffected SCCs.

The incremental verification results from [59] are extended in [39] to both model construction and quantitative verification, two important stages in probabilistic model checking. Model construction represents the synthesis of an MDP from a high-level modelling language such as the PRISM language used to specify the model in Fig. 1b. The incremental approach to model construction involves finding the set of states to be rebuilt after a change, which reduces the set of high-level modelling commands to be evaluated. For quantitative verification [39] decomposes the system model into SCCs as in [59]. However, [39] uses policy iteration to establish the correctness of a formula, while [59] applies value iteration. The advantage of using policy iteration incrementally is the capability to reuse policies between verification runs and to specify the MDP *adversary* (i.e., resolution of the MDP nondeterminism) with which the computation starts. This could reduce the number of iterations due to faster convergence. Intuitively, a good initial adversary is the optimal adversary from the previous execution. Another improvement over [59] is the support for small changes in the structure of the analysed model. For additional information see [38].

The framework for syntax-driven incremental verification from [6] requires the specification of the structure of a software system using operator precedence grammars. These grammars natively support incremental parsing, which enables the synthesis of incremental verification procedures. When changes occur, the approach employs incremental algorithms for traversing and evaluating the part

of the syntax tree affected by the changes. Preliminary results from the application of the approach to the quantitative verification of reliability properties are very promising [5].

Finally, the Δ *evaluation* introduced by Meedeniya and Grunske [60] addresses the incremental verification of reliability requirements in component-based systems. The behaviour of the analysed system is modelled as a DTMC, where each state of the DTMC is associated with a system component as suggested in [24]. When a component is in control of the execution, it can either transfer control to another component (according to a workflow), or transition to an absorbing state denoting a failed or successful execution. When a single change occurs in the model, using operations from matrix theory, it is possible to analyse the impact of the change and re-evaluate the reliability of the system without a complete re-evaluation. The approach is applicable to scenarios that involve the analysis of reliability requirements for systems affected by a single component change at a time.

Parametric RQV. Self-adaptive systems are characterised by uncertainty due to factors such as incomplete system specification at design time and runtime changes in the environment and the system itself. *Parametric Markov models* support the specification of this uncertainty for the system under consideration. In these models, some transition probabilities are not fixed, but are associated with parameters whose values are unknown until runtime and may change during the system operation. Using a process that comprises design time and runtime steps, it is possible to reason about the satisfaction of QoS requirements with low runtime overheads. At design time, QoS requirements are translated into algebraic expressions in a (computationally expensive) pre-computation step. At runtime, these algebraic expressions are evaluated by replacing the unknown parameters with the actual values obtained through system monitoring. This runtime evaluation step takes a fraction of the time required to carry out quantitative verification on the actual model. The approach is called parametric model checking [26]. The design time, pre-computation step used to derive the algebraic expressions evaluated at runtime is described in more detail below.

Daws' parametric model checking work [26] introduces a language-theoretic approach to symbolic probabilistic model checking of reachability properties over DTMCs. The approach initially converts a DTMC into a finite state automaton in which transition probabilities are modelled as letters of an alphabet, followed by the synthesis of a regular expression that defines the language recognised by the automaton using state elimination algorithms. This regular expression then undergoes a recursive evaluation that yields a rational algebraic expression for the property to evaluate. The approach works well for reachability properties, but lacks support for other types of probabilistic temporal logic properties, including reward properties like those from Example 1. Another limitation of the approach is related to the length of the regular expressions it operates with, which can grow to $n^{\Theta(\log n)}$ in the worst-case scenario, where n represents the number of states in the model.

In [45, 48], the authors draw upon the work presented by Daws [26] to devise an effective approach that combines state elimination with early evaluation of the rational function. In each iteration of the approach, a state elimination step is followed by on-the-fly simplification of the rational function taking advantage of cancellations, symmetries and simplifications of arithmetic expressions. Compared to [26], the algorithm requires n^3 operations in most cases, which represents a significant improvement. However, in the worst case the length of the rational function is still $n^{\Theta(\log n)}$. This can occur if no rational function can be simplified during the entire process, which is a uncommon scenario according to the findings in [45, 48]. The approach underpins the operation of the model checker PARAM [47] and is also implemented in PRISM [56].

An approach similar to [26, 45, 48] is proposed in [35]. In this approach, the computation time required to derive the algebraic expression for a set of reliability-related QoS requirements and a parametric DTMC depends on the model size, the number of parametric states and the number of outgoing transitions from these states. The comparison of the approach with the probabilistic model checkers PRISM [56] and MRMC [52] showed that the time taken by the runtime step of the approach is several orders of magnitude faster than both probabilistic model checkers. An extension of the approach supporting the derivation of algebraic expressions for DTMCs augmented with cost/reward structures is presented in [34]. For an extended version of the works in [34, 35], see [37].

The work by Hahn et al. [46] takes a different perspective and considers the problem of parameter synthesis of PCTL formulae for parametric models. Instead of generating a rational function that represents a reachability requirement, the approach synthesises the set of parameter values for which the reachability requirement holds. At design time, applying recursively state space exploration techniques, the parameter space is partitioned into hyper-rectangles, i.e., regions in the dimension of the model parameters that represent families of models. Each of these regions provides globally the same output, that is, the requirement holds (or not) for all the concrete models resulting from instantiations of the parameters with values in this region. Note that the approach allows a limited state space area to remain unknown; evaluation in this area is very complex and is left undecided. When the system undergoes changes at runtime, it is sufficient to access these hyper-rectangles and instantaneously assess whether the requirement is still satisfied or not. A preliminary implementation of the approach has been developed as part of PARAM [47].

The approaches presented in this section achieve significant improvements in runtime quantitative verification both in terms of computation time and memory consumption. The computationally expensive model exploration is carried out only once at design time, while the runtime step involves the evaluation of a set of algebraic expressions in [26, 34, 35, 45, 48] or quickly accessing a lookup table in [46]. Enhancing further these approaches to deal with a larger number of parameters and with structural model changes is the subject of current research.

Conventional Software Engineering Methods. The work in [40] extends the applicability of established efficiency improvement techniques to runtime quantitative verification. Firstly, the *caching* of recent verification enables their reuse and reduces the number of runtime reverification steps when system changes are small and localised. Secondly, *limited lookahead* uses idle CPU cycles to pre-verify states of the system that are likely to occur in the near future. When system changes arise, if the current system state has already been verified, it is sufficient to retrieve these verification results. Finally, *nearly-optimal reconfiguration* terminates the search for the next configuration of a self-adaptation system as soon as a valid configuration has been identified and a “near optimality” stopping criterion is met.

These techniques were evaluated using a simulator for self-adaptive unmanned underwater vehicles. The findings reported in [40] show that all the techniques (and their combinations) improve the response time in many scenarios encountered in practice. At the same time, each technique is associated with extra costs. Thus, caching and limited lookahead require additional storage for the recent verification results, while the latter technique also needs additional CPU for the pre-verification process. In contrast, nearly-optimal reconfiguration operates with insignificant overheads compared to the standard RQV, at the expense of selecting a sub-optimal configuration.

Search-based software engineering [49] is another conventional method that has the potential to reduce the overheads of RQV and/or to increase its scalability. The recent work in [41] demonstrates that this method can generate effective new configurations using large stochastic models at design time, and the extension of this solution for exploitation in a runtime context is worth exploring.

Decentralised RQV. Self-adaptive systems are becoming increasingly complex and are often distributed. Accordingly, recent research roadmaps have advocated the use of decentralised control in self-adaptive systems [27, 69] as a way to eliminate the single point of failure associated with centralised control loops, to improve the efficiency and scalability of self-adaptation, and to realise the original autonomic computing vision [53]. To this end, Weyns et al. [70] propose several patterns of interacting control loops for the engineering of decentralised control in self-adaptive systems, while [61, 66, 68] present promising heuristics from this research area.

Motivated by these efforts, the work in [8] has introduced the first approach to using RQV for the engineering of decentralised control loops in distributed self-adaptive systems. In this approach, the system components provided with a self-adaptation control loop execute a four-stage decentralised control workflow:

1. In a *local capability analysis* stage, the RQV of component-level models determines alternative contributions that the component can make towards satisfying the system-level QoS requirements. These possible contributions are used to assemble a local *capability summary* for the component.
2. In the *capability summary exchange* stage, the components share their local capability summaries with the peer components. This stage is executed only

- on start-up and after infrequent changes that invalidate the local capability summaries.
3. In the *contribution-level agreement (CLA) selection* stage, each component independently executes a planning algorithm that selects the contribution it should make towards achieving the system-level QoS requirements. This is one of the alternative contributions from the local capability summary of the component. The selection method guarantees that the system-level QoS requirements are met as long as all components “agree” to achieve the local contributions selected in this way.
 4. Finally, the *local control* stage uses RQV-driven self-adaptation to ensure that each component meets its CLA. The method used to establish the CLAs makes this possible at most times. The infrequent situations when the CLA cannot be achieved locally are identified by the local control loop, and triggers a re-execution of the local capability analysis.

The approach was evaluated in a case study from the unmanned underwater vehicles domain [8], with promising results. Thus, the distributed UUV system simulated in the case study achieved its QoS requirements with similar success rates when using decentralised and centralised RQV-based self-adaptation. However, the RQV overheads for the decentralised approach were several orders of magnitude lower than those for the centralised RQV solution. Additionally, the decentralised approach scaled to much larger system sizes, with acceptably low communication rates for the UUV domain, and without introducing a single point of failure. The only downside of the decentralised approach is that components need to be conservative in their local capability analysis, so that their local control loops can achieve the selected CLAs most of the time despite local changes. This typically leads to a reduction in the utility with which the distributed system operates—see Example 2 for details about the use of utility functions in RQV-driven self-adaptive systems.

3.2 Learning Probabilistic Models

The ability of runtime quantitative verification to provide assurance evidence for self-adaptive systems depends on the accuracy of the models analysed by the technique. Inaccurate models may lead to invalid reasoning about the compliance of a system with its QoS requirements, and thus to incorrect adaptation decisions. To avoid this, the initial parametric model from Fig. 2 is typically developed by someone with probabilistic modelling expertise. In addition, the techniques used to learn the model parameters from observations of the system behaviour must synthesise accurate values for these parameters. The two parts of this section describe several of these techniques, and mention a number of approaches to learning the structure of Markov models, respectively.

Parameter Learning. In [33], Epifani et al. introduce a Bayesian technique for learning the unknown state transition probabilities of a DTMC. The technique

uses observations of all outgoing transitions from the states with unknown transition probabilities, and an *a priori* estimate for each of these transitions. The relative contributions of the observations and of the *a priori* estimate is specified by a *smoothing parameter*, with large values of this parameter placing more confidence in the *a priori* estimate but slowing down the learning process. The experimental results from [33] demonstrate that the technique produces accurate results given a sufficiently large number of observations. A detailed analysis of the technique is available in [36].

The work in [13] extends the applicability of the Bayesian learning technique proposed by Epifani et al. [33] to scenarios in which the unknown transitional probabilities vary significantly over time. To this end, the contribution of each observation of a state transition to the calculation of transition probability estimates decreases exponentially with the age of the observation. This ensures that older observations, which may correspond to parameter values that are no longer relevant, have little impact on the parameter estimates. The downside of the approach is that more frequent observations than in [33] are required to achieve smooth learning despite old observations being partly discarded. Varying the *ageing coefficient*, i.e., the rate with which observations are “forgotten” as they grow older, reduces this undesired effect.

The results from [13, 33] are further improved in [19], where the authors propose an adaptive approach to learning the unknown transition probabilities of parametric DTMCs. This approach works by dynamically updating the smoothing parameter and the ageing coefficient introduced in [33] and [13], respectively, based on the frequency of the observations of state transitions.

Model Learning. The parameter learning techniques described above can be used in scenarios in which the structure of the analysed model is known in advance and does not change over time. This assumption does not hold for certain types of self-adaptive systems. For instance, the failure of system components may require the use of alternative combination of operations to achieve the functional requirements of the system. Likewise, users may interact with self-adaptive systems in ways not anticipated at design time, and therefore not captured by the structure of the initial models of these systems. Learning the structure of probabilistic models with acceptable accuracy requires large numbers of observations of the system behaviour. Therefore, only offline approaches to learning probabilistic models have been proposed so far. These include approaches for learning continuous-time Markov chains [65], labelled probabilistic transition systems [54] and discrete-time Markov chains [44] from execution traces, tree samples and application logs, respectively.

4 Applications

Although runtime quantitative verification is a relatively new approach to providing assurance evidence, it has already been applied to self-adaptive systems from multiple domains, including:

- service-based systems [11, 13, 19, 33, 36, 44, 67];
- cloud computing [17, 18, 51, 62];
- embedded and robotic systems [8, 31, 40, 71];
- multi-tier software architectures [20–22];
- dynamic software product lines [32];
- dynamic power management [18].

This section summarises applications from the first two of these domains.

4.1 Self-Verifying Service-Based Systems

Built through the integration of third-party services accessed over the Internet, service-based systems (SBSs) are increasingly used in business-critical and safety-critical applications where they must comply with strict QoS requirements. Self-adaptive SBSs achieve this compliance by selecting the *concrete services* for their operations dynamically, from sets of functionally equivalent third-party services with different levels of performance, reliability and cost. Figure 4 depicts the architecture of a *self-verifying service-based system* in which this concrete service selection is underpinned by RQV.

Originally proposed in [11] and further extended in [10, 14, 15], the self-adaptive SBS architecture from Fig. 4 comprises $n \geq 1$ operations performed by remote third-party services. The $n \geq 1$ COVE proxies in this architecture interface the SBS workflow with sets of remote service such that the i -th SBS operation can be carried out by $m_i \geq 1$ functionally equivalent services. The main role of the proxies is to ensure that each execution of an SBS operation is carried out through the invocation of a concrete service selected as described below. Whenever an instance of the i -th proxy is created, it is initialised with a sequence of “promised” service-level agreements $sla_{ij} = (p_{ij}^0, c_{ij})$, $1 \leq j \leq m_i$, where $p_{ij}^0 \in [0, 1]$ and $c_{i,j} > 0$ represent the provider-supplied probability of success and the cost for an invocation of service $s_{i,j}$, respectively.

The n proxies are also responsible for notifying a *model updater* about each service invocation and its outcome. The model updater starts from a developer-supplied initial Markovian model of the SBS workflow, and uses the online learning techniques from Sect. 3.2 to adjust the transition probabilities of the model in line with these proxy notifications. Finally, the up-to-date Markovian model maintained by the model updater is used by an *autonomic manager* that performs RQV to select the service combination used by the n proxies so that it satisfies the SBS requirements with minimal cost at all times. Accordingly, the proxies, model updater and autonomic manager with its quantitative verification engine implement the RQV control loop from Fig. 2.

4.2 Dynamic Management of Cloud Computing Infrastructure

Cloud computing providers offer infrastructure, platforms and software to their customers on a pay-as-you go basis. Software-as-a-Service is an attractive option for customers as it places software installation and maintenance responsibilities

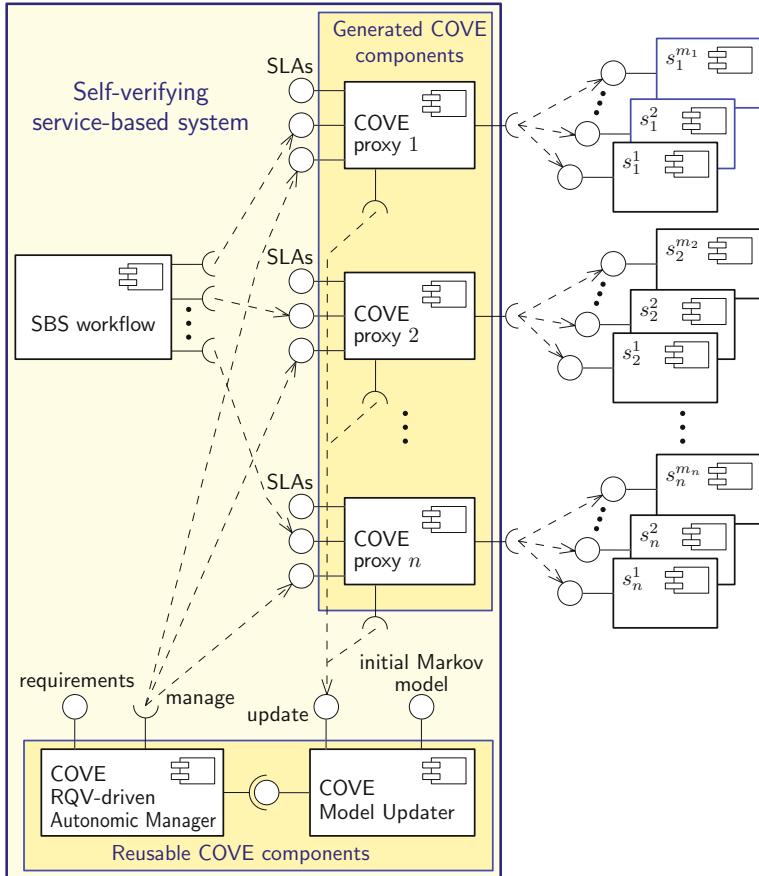


Fig. 4. Self-verifying SBS with RQV-driven selection of concrete services, implemented using the tool-supported continual verification framework COVE (<http://www-users.cs.york.ac.uk/~raduc/COVE/>) [14]

with the cloud provider. A software service is typically implemented by the provider as a multi-tier architecture. For example, Fig. 5 depicts the deployment of a service that comprises three *functions*: web, application and database. Several instances of each function run on different virtual machines (VMs), which are deployed across four physical servers that make up the cloud infrastructure.

Services deployed on cloud infrastructure take advantage of its elastic nature, scaling rapidly to meet demands by adding new servers, virtual machines and function instances. The provider can alter service availability according to changes in demand, costs of service provision or according to reliability criteria set by SLAs with the customer. These types of changes are governed by policies and are often carried out programmatically in response to fluctuations of the service workload. Other changes to the service happen unexpectedly. Examples

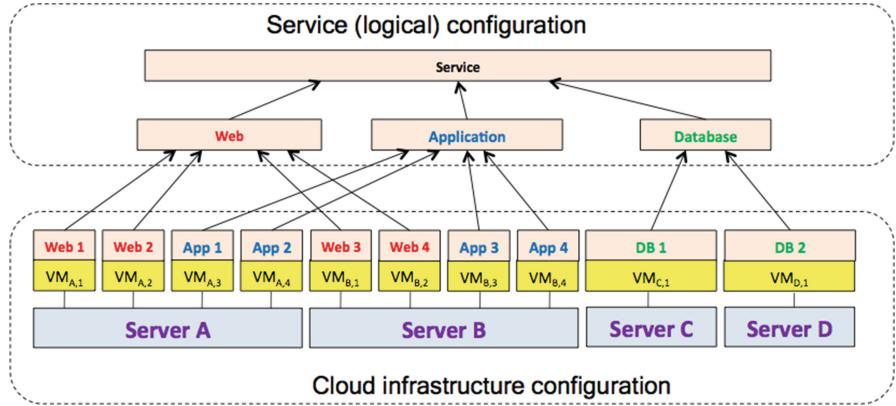


Fig. 5. Three-tier architecture of a cloud-deployed service

of unexpected changes include hardware failure of physical servers or failure of function instances arising from software faults. By applying RQV, datacentre administrators can gain valuable insight into the impact that each change has on the service reliability by obtaining precise answers to questions such as

- Q1 What is the maximum probability of a service failing¹ over a one-month time period?
- Q2 How will the probability of failure for a service be affected if one of its database instances is switched off to reflect a decrease in service workload?
- Q3 How many additional VMs should be used to run the instances of a service function when these instances are moved to VMs placed on physical servers with fewer/less reliable memory blocks?

Cloud datacentre administrators or their automated resource management scripts then have the option of reacting with remedial action if the service fails to meet its QoS requirements. They can also discard planned changes (e.g., a removal of a function instance) whose implementation would violate these requirements.

Modelling Cloud-Deployed Services. To apply RQV to the service from Fig. 5, we devised [17] the following probabilistic automata (PAs) that formalise the behaviour of the system components:

- Four server PAs (A, B, C and D) that model the behaviour of the four physical servers;
- Two PAs ($webapp_A, webapp_B$) that model the web and application function instances deployed on servers A and B, and two further PAs (db_C, db_D) that model the database function instances on servers C and D.
- A single service PA model $service$ that models the multi-tier service.

¹ A service failed if all instances of one of its functions are unavailable.

Standard quantitative verification requires the monolithic model

$$M = A \parallel B \parallel C \parallel D \parallel \text{webapp}_A \parallel \text{webapp}_B \parallel \text{db}_C \parallel \text{db}_D \parallel \text{service} \quad (2)$$

to be constructed from the parallel composition of all component models in order to analyse the interleaving and synchronised behaviour between components. Despite significant progress in improving the efficiency of quantitative verification techniques and tools, using the model from (2) often results in a high overhead or intractable verification task, even for small software services.

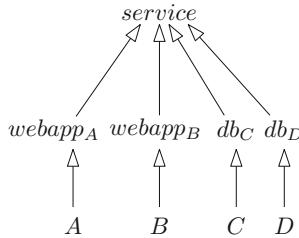


Fig. 6. Dependencies between the component models of the cloud service

Compositional and Incremental RQV of Cloud-Deployed Systems. To avoid the state explosion problem, the compositional and incremental RQV techniques described in Sect. 3.1 have been applied in [17, 51] by taking advantage of the modular structure of cloud-deployed services.

These techniques perform verification tasks component-wise under *assumptions* that encode component properties and restrict the possible behaviour of other, dependent system components. More specifically, the interdependencies between the models of the hardware, software and other components of the cloud service are derived from the architecture in Fig. 5, and are expressed as a dependency tree (Fig. 6). By associating component models with probabilistic safety properties, probabilistic assume-guarantee rules (1) are applied to models in the order specified by a depth-first traversal of this dependency tree.

While this compositional analysis broadens the range of systems to which RQV can be applied effectively, continual service changes require frequent runtime reverification. We describe two change scenarios typically encountered by an administrator of a cloud-deployed service with the dependency tree in Fig. 6, and present at a high level the reverification steps performed by incremental RQV to establish the impact of the changes from these scenarios. The two scenarios are described in detail in [51], and assume that the service has already been verified compositionally as described above.

Scenario 1: In response to an increase in service workload, the administrator deploys a new instance of the database on physical server C. This results in an updated component model db'_C modelling two database instances. The

sequence of models that may need reverification is $\langle db'_C, service \rangle$, since the *service* model uses assumptions obtained from the verification of db'_C , according to the dependencies in Fig. 6. It is sometimes the case that changes in a component result in improved reliability and satisfy local QoS properties that allow the reverification to terminate early. In this scenario, the reliability of the database instances on C is improved with the addition of a new database instance, and the reverification process halts without the need to reverify the *service* model.

Scenario 2: Suppose that physical server A has an unexpected failure in one of its hard disks, causing a degradation in reliability. This change results in an updated component model A' that reflects the reduction in the number of operational hard disks. The component models from the sequence $\langle A', webapp_A, service \rangle$ may need reverification in this scenario. The model *webapp_A* requires reverification because its verification uses the reliability assumptions of A' . Furthermore, because the reverification of *webapp_A* is required, the *service* model must also be included in the reverification sequence, so that the impact on the reliability QoS properties of the entire service may be determined.

The experiments carried out in [17, 51] show that compositional and incremental RQV provide significant reductions in the time required to obtain assurance evidence supporting the self-adaptation of cloud-deployed services.

5 Summary and Research Challenges

We described runtime quantitative verification (RQV), a formal verification technique used to provide assurance evidence that self-adaptive software meets its dependability, performance and other QoS requirements. RQV has been successfully evaluated in case studies involving the self-adaptation of software systems including service-based, cloud-deployed and embedded systems. We introduced RQV using a running example from the area of unmanned underwater exploration, and we summarised two recent case studies from the literature.

RQV requires the continual learning and quantitative verification of probabilistic models of the self-adaptive software—two tasks that pose significant research challenges. Addressing these challenges has been the objective of considerable research within the past several years. As a result, new RQV techniques have emerged that are capable of verifying and reverifying the probabilistic models of self-adaptive systems with low overheads and much faster than standard quantitative verification. We overviewed the state-of-the-art RQV techniques devised by this research effort, indicating their applications and limitations. Complementary research summarised in the paper has tackled challenges associated with the runtime learning and updating of probabilistic models from observations of the behaviour of the modelled software.

Notwithstanding these significant advances, much remains to be done to improve the applicability and to ease the adoption of RQV. The main research

challenges in this area arise from the need to carry out complex software and systems engineering activities continually, in the presence of uncertainty and with limited or no human supervision. Several of these research challenges are summarised below.

Continual Modelling. The ability of RQV to provide assurance evidence depends on the accuracy of the probabilistic models it operates with. Additional research is required to ensure that these models are continually updated to reflect the evolving structure, parameters and environment of self-adaptive systems. Furthermore, the models analysed by RQV are approximations of the actual behaviour of self-adaptive systems, and new research is needed to quantify the accuracy of these approximations.

Continual Verification. More needs to be done to extend the applicability of efficient RQV techniques such as compositional, incremental and parametric model checking to additional types of models, in particular to continuous-time models. To use RQV results as assurance evidence, we need to be able to formally quantify the confidence that can be placed in these results. New RQV techniques capable of providing this quantification, e.g. by extending recent results on quantitative verification with confidence intervals [9, 12], are required.

Continual Assurance. The provision of assurance evidence by RQV is only one stage of the assurance process. Exploiting this assurance evidence will require its integration with evidence from other sources (e.g., testing and design-time verification). Furthermore, RQV will have to be integrated with the modelling and analysis methods used by the other stages of the assurance process. Last but not least, the assurance process itself will need to be extended in order to support the continual assurance required for self-adaptive systems.

Integration with Discrete-Event Control. Self-adaptive software used in safety-critical and business-critical applications must meet both QoS and functional requirements. RQV is particularly suitable for achieving compliance with QoS requirements, while complementary approaches based primarily on discrete-event control (e.g. [29, 30]) are suited for ensuring functional compliance. The integration of the two approaches is necessary [1] in order to provide assurances that self-adaptive software simultaneously meets both types of requirements.

References

1. Alur, R., Henzinger, T.A., Vardi, M.Y.: Theory in practice for system design and verification. ACM SIGLOG News **2**(1), 46–51 (2015)
2. Ardagna, D., Pernici, B.: Adaptive service composition in flexible processes. IEEE Trans. Softw. Eng. **33**(6), 369–384 (2007)

3. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model-checking continuous-time Markov chains. *ACM Trans. Comput. Logic* **1**(1), 162–170 (2000)
4. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press, Cambridge (2008)
5. Bianculli, D., Filieri, A., Ghezzi, C., Mandrioli, D.: A syntactic-semantic approach to incremental verification. *CoRR*, abs/1304.8034 (2013)
6. Bianculli, D., Filieri, A., Ghezzi, C., Mandrioli, D.: Syntactic-semantic incrementality for agile verification. *Sci. Comput. Program.* **97**(1), 47–54 (2015)
7. Calinescu, R.: Emerging techniques for the engineering of self-adaptive high-integrity software. In: Cámaras, J., de Lemos, R., Ghezzi, C., Lopes, A. (eds.) Assurances for Self-Adaptive Systems: Principles, Models, and Techniques. LNCS, vol. 7740, pp. 297–310. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36249-1_11
8. Calinescu, R., Gerasimou, S., Banks, A.: Self-adaptive software with decentralised control loops. In: Egyed, A., Schaefer, I. (eds.) FASE 2015. LNCS, vol. 9033, pp. 235–251. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46675-9_16
9. Calinescu, R., Ghezzi, C., Johnson, K., Pezze, M., Rafiq, Y., Tamburrelli, G.: Formal verification with confidence intervals to establish quality of service properties of software systems. *IEEE Trans. Reliab.* **PP**(99), 1–19 (2015)
10. Calinescu, R., Ghezzi, C., Kwiatkowska, M., Mirandola, R.: Self-adaptive software needs quantitative verification at runtime. *Commun. ACM* **55**(9), 69–77 (2012)
11. Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G.: Dynamic QoS management and optimization in service-based systems. *IEEE Trans. Softw. Eng.* **37**(3), 387–409 (2011)
12. Calinescu, R., Johnson, K., Paterson, C.: FACT: a probabilistic model checker for formal verification with confidence intervals. In: 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2016)
13. Calinescu, R., Johnson, K., Rafiq, Y.: Using observation ageing to improve Markovian model learning in QoS engineering. In: 2nd ACM/SPEC International Conference on Performance Engineering (ICPE 2011), pp. 505–510 (2011)
14. Calinescu, R., Johnson, K., Rafiq, Y.: Developing self-verifying service-based systems. In: 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013), pp. 734–737 (2013)
15. Calinescu, R., Johnson, K., Rafiq, Y.: Using continual verification to automate service selection in service-based systems. Technical report YCS-2013-484, Department of Computer Science, University of York (2013). <http://www.cs.york.ac.uk/ftpdir/reports/2013/YCS/484/YCS-2013-484.pdf>
16. Calinescu, R., Kikuchi, S.: Formal methods @ runtime. In: Calinescu, R., Jackson, E. (eds.) Monterey Workshop 2010. LNCS, vol. 6662, pp. 122–135. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21292-5_7
17. Calinescu, R., Kikuchi, S., Johnson, K.: Compositional reverification of probabilistic safety properties for large-scale complex IT systems. In: Calinescu, R., Garlan, D. (eds.) Monterey Workshop 2012. LNCS, vol. 7539, pp. 303–329. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34059-8_16
18. Calinescu, R., Kwiatkowska, M.: Using quantitative analysis to implement autonomic IT systems. In: 31st IEEE International Conference on Software Engineering (ICSE 2009), pp. 100–110 (2009)

19. Calinescu, R., Rafiq, Y., Johnson, K., Bakir, M.E.: Adaptive model learning for continual verification of non-functional properties. In: 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014), pp. 87–98 (2014)
20. Cámera, J., de Lemos, R.: Evaluation of resilience in self-adaptive systems using probabilistic model-checking. In: 2012 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 53–62, June 2012
21. Cámera, J., Garlan, D., Schmerl, B., Pandey, A.: Optimal planning for architecture-based self-adaptation via model checking of stochastic games. In: Proceedings of the 10th DADS Track of the 30th ACM Symposium on Applied Computing, Salamanca, Spain, 13–17 April 2015
22. Cámera, J., Moreno, G.A., Garlan, D.: Stochastic game analysis and latency awareness for proactive self-adaptation. In: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, pp. 155–164. ACM, New York (2014)
23. Cardellini, V., Casalicchio, E., Grassi, V., Iannucci, S., Lo Presti, F., Mirandola, R.: Moses: a framework for QoS driven runtime adaptation of service-oriented systems. *IEEE Trans. Softw. Eng.* **38**(5), 1138–1159 (2012)
24. Cheung, R.: A user-oriented software reliability model. *IEEE Trans. Softw. Eng.* **6**(2), 118–125 (1980)
25. Coppit, D., Yang, J., Khurshid, S., Le, W., Sullivan, K.: Software assurance by bounded exhaustive testing. *IEEE Trans. Softw. Eng.* **31**(4), 328–339 (2005)
26. Daws, C.: Symbolic and parametric model checking of discrete-time Markov chains. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 280–294. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31862-0_21
27. de Lemos, R., et al.: Software engineering for self-adaptive systems: a second research roadmap. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 1–32. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_1
28. D’Ippolito, N., Braberman, V., Kramer, J., Magee, J., Sykes, D., Uchitel, S.: Hope for the best, prepare for the worst: multi-tier control for adaptive systems. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp. 688–699. ACM, New York (2014)
29. D’Ippolito, N., Braberman, V., Piterman, N., Uchitel, S.: Synthesis of live behaviour models for fallible domains. In: 33rd International Conference on Software Engineering (ICSE), pp. 211–220, May 2011
30. D’Ippolito, N.R., Braberman, V., Piterman, N., Uchitel, S.: Synthesis of live behaviour models. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2010, pp. 77–86. ACM, New York (2010)
31. Dräger, K., Forejt, V., Kwiatkowska, M., Parker, D., Ujma, M.: Permissive controller synthesis for probabilistic systems. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 531–546. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_44
32. Dubslaff, C., Klüppelholz, S., Baier, C.: Probabilistic model checking for energy analysis in software product lines. In: Proceedings of the 13th International Conference on Modularity, MODULARITY 2014, pp. 169–180. ACM, New York (2014)
33. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by runtime parameter adaptation. In: 31st IEEE International Conference on Software Engineering (ICSE 2009), pp. 111–121 (2009)

34. Filieri, A., Ghezzi, C.: Further steps towards efficient runtime verification: handling probabilistic cost models. In: Formal Methods Software Engineering: Rigorous and Agile Approaches (FormSERA 2012), pp. 2–8 (2012)
35. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: 33rd International Conference on Software Engineering (ICSE 2011), pp. 341–350 (2011)
36. Filieri, A., Ghezzi, C., Tamburrelli, G.: A formal approach to adaptive software: continuous assurance of non-functional requirements. *Formal Aspects Comput.* **24**(2), 163–186 (2012)
37. Filieri, A., Tamburrelli, G.: Probabilistic verification at runtime for self-adaptive systems. In: Cámaras, J., de Lemos, R., Ghezzi, C., Lopes, A. (eds.) Assurances for Self-Adaptive Systems: Principles, Models, and Techniques. LNCS, vol. 7740, pp. 30–59. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36249-1_2
38. Forejt, V., Kwiatkowska, M., Parker, D., Qu, H., Ujma, M.: Incremental runtime verification of probabilistic systems. Technical report RR-12-05, Department of Computer Science, University of Oxford (2012)
39. Forejt, V., Kwiatkowska, M., Parker, D., Qu, H., Ujma, M.: Incremental runtime verification of probabilistic systems. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 314–319. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35632-2_30
40. Gerasimou, S., Calinescu, R., Banks, A.: Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration. In: 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2014), pp. 115–124 (2014)
41. Gerasimou, S., Tamburrelli, G., Calinescu, R.: Search-based synthesis of probabilistic models for quality-of-service software engineering. In: 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 319–330, November 2015
42. Ghezzi, C.: Evolution, adaptation, and the quest for incrementality. In: Calinescu, R., Garlan, D. (eds.) Monterey Workshop 2012. LNCS, vol. 7539, pp. 369–379. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34059-8_19
43. Ghezzi, C., Greenyer, J., La Manna, V.P.: Synthesizing dynamically updating controllers from changes in scenario-based specifications. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2012, pp. 145–154. IEEE Press, Piscataway (2012)
44. Ghezzi, C., Pezzè, M., Sama, M., Tamburrelli, G.: Mining behavior models from user-intensive web applications. In: 36th International Conference on Software Engineering (ICSE 2014), pp. 277–287 (2014)
45. Hahn, E., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric Markov models. *Int. J. Softw. Tools Technol. Transfer* **13**(1), 3–19 (2011)
46. Hahn, E.M., Han, T., Zhang, L.: Synthesis for PCTL in parametric Markov decision processes. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 146–161. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_12
47. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: PARAM: a model checker for parametric Markov models. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 660–664. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_56
48. Hahn, E.M., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric Markov models. In: Păsăreanu, C.S. (ed.) SPIN 2009. LNCS, vol. 5578, pp. 88–106. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02652-2_10

49. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: trends, techniques and applications. *ACM Comput. Surv.* **45**(1), 11:1–11:61 (2012)
50. Hoare, C.A.R.: An axiomatic basis for computer programming. *Commun. ACM* **12**(10), 576–580 (1969)
51. Johnson, K., Calinescu, R., Kikuchi, S.: An incremental verification framework for component-based software systems. In: 16th International Symposium on Component-Based Software Engineering (CBSE 2013), pp. 33–42 (2013)
52. Katoen, J.-P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. *Perform. Eval.* **68**(2), 90–104 (2011)
53. Kephart, J., Chess, D.: The vision of autonomic computing. *Computer* **36**(1), 41–50 (2003)
54. Komuravelli, A., Pasareanu, C.S., Clarke, E.M.: Learning probabilistic systems from tree samples. In: 27th IEEE/ACM Symposium on Logic in Computer Science (LICS 2012), pp. 441–450 (2012)
55. Kwiatkowska, M.: Quantitative verification: models, techniques and tools. In: 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007), pp. 449–458 (2007)
56. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
57. Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Assume-guarantee verification for probabilistic systems. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 23–37. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_3
58. Kwiatkowska, M., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. *Formal Methods Syst. Des.* **29**(1), 33–78 (2006)
59. Kwiatkowska, M., Parker, D., Qu, H.: Incremental quantitative verification for Markov decision processes. In: 41st IEEE/IFIP International Conference on Dependable Systems Networks (DSN 2011), pp. 359–370 (2011)
60. Meedeniya, I., Grunske, L.: An efficient method for architecture-based reliability evaluation for evolving systems with changing parameters. In: 21st IEEE International Symposium on Software Reliability Engineering (ISSRE 2010), pp. 229–238 (2010)
61. Nallur, V., Bahsoon, R.: A decentralized self-adaptation mechanism for service-based applications in the cloud. *IEEE Trans. Softw. Eng.* **39**(5), 591–612 (2013)
62. Naskos, A., Stachtiari, E., Gounaris, A., Katsaros, P., Tsoumakos, D., Konstantinou, I., Sioutas, S.: Dependable horizontal scaling based on probabilistic model checking. In: 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2015) (2015)
63. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems, pp. 123–144. Springer, New York (1985). https://doi.org/10.1007/978-3-642-82453-1_5
64. Segala, R., Lynch, N.A.: Probabilistic simulations for probabilistic processes. *Nord. J. Comput.* **2**(2), 250–273 (1995)
65. Sen, K., Viswanathan, M., Agha, G.: Learning continuous time Markov chains from sample executions. In: Quantitative Evaluation of Systems, pp. 146–155 (2004)

66. Sykes, D., Magee, J., Kramer, J.: Flashmob: distributed adaptive self-assembly. In: 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011), pp. 100–109 (2011)
67. Weyns, D., Calinescu, R.: Tele assistance system: an exemplar for self-adaptive service-based systems. In: 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015) (2015, to appear)
68. Weyns, D., Haesvoets, R., Helleboogh, A., Holvoet, T., Joosen, W.: The MACODO middleware for context-driven dynamic agent organizations. ACM Trans. Auton. Adapt. Syst. **5**(1), 3:1–3:28 (2010)
69. Weyns, D., Malek, S., Andersson, J.: On decentralized self-adaptation: lessons from the trenches and challenges for the future. In: 5th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010), pp. 84–93 (2010)
70. Weyns, D., et al.: On patterns for decentralized control in self-adaptive systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 76–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_4
71. Wongpiromsarn, T., Ulusoy, A., Belta, C., Frazzoli, E., Rus, D.: Incremental synthesis of control policies for heterogeneous multi-agent systems with linear temporal logic specifications. In: 2013 IEEE International Conference on Robotics and Automation (ICRA), pp. 5011–5018, May 2013
72. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: practice and experience. ACM Comput. Surv. **41**(4), 19:1–19:36 (2009)
73. Younes, H.L.S.: Ymer: a statistical model checker. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 429–433. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_43
74. Zeng, L., Benatallah, B., Ngu, A., Dumas, M., Kalagnanam, J., Chang, H.: QoS-aware middleware for web services composition. IEEE Trans. Softw. Eng. **30**(5), 311–327 (2004)

Integration and Coordination

Contracts-Based Control Integration into Software Systems

Filip Křikava¹(✉), Philippe Collet³, Romain Rouvoy², and Lionel Seinturier²

¹ Czech Technical University, Prague, Czech Republic

filip.krikava@fit.cvut.cz

² University of Lille/Inria, Lille, France

{romain.rouvoy,lionel.seinturier}@inria.fr

³ Université Nice Sophia Antipolis, Nice, France

philippe.collet@unice.fr

Abstract. Among the different techniques that are used to design self-adaptive software systems, control theory allows one to design an adaptation policy whose properties, such as stability and accuracy, can be formally guaranteed under certain assumptions. However, in the case of software systems, the integration of these controllers to build complete feedback control loops remains manual. More importantly, it requires an extensive handcrafting of non-trivial implementation code. This may lead to inconsistencies and instabilities as no systematic and automated assurance can be obtained on the fact that the initial assumptions for the designed controller still hold in the resulting system.

In this chapter, we rely on the principles of design-by-contract to ensure the correction and robustness of a self-adaptive software system built using feedback control loops. Our solution raises the level of abstraction upon which the loops are specified by allowing one to define and automatically verify system-level properties organized in contracts. They cover behavioral, structural and temporal architectural constraints as well as explicit interaction. These contracts are complemented by a first-class support for systematic fault handling. As a result, assumptions about the system operation conditions become more explicit and verifiable in a systematic way.

1 Introduction

Self-Adaptive Software Systems (SASS) are characterized by the ability to continuously operate under varying runtime conditions [11]. They autonomously adapt themselves based on the perception of both their own state and the state of their environment. The heart of this adaptation capability can be based on the notion of a *Feedback Control Loop* (FCL) that regulates the characteristics of the system to achieve its goals despite changes that may occur during operation [20]. In particular, it uses measurements of system outputs (*e.g.* response time) to automatically adjusts system control inputs (*e.g.* level of concurrency) based on a given control strategy.

There are different techniques that can be used to design SASS. Among them, control theory offers a promising solution by providing a well-established mathematical foundations for designing controllers. It has been extensively used in other engineering disciplines for controlling behavior of industrial plants. It allows one to develop a control strategy whose properties, such as stability and accuracy, can be formally guaranteed under certain *assumptions on the operating conditions* [17] such as input data bounds or timing properties.

However, there are important difficulties in systematically applying control techniques into software systems [17]. First, it is quite hard for non-experts to develop mathematical models of software behavior that are actually usable for a control design. Next, the design, implementation and integration of the controller into a complete self-adaptive software system are activities known to be both error-prone and difficult [21]. Despite support provided by tools, such as MATLAB, SIMULINK, or SYSWEAVER [32], that provide code generation capabilities, the controller integration still requires an extensive handcrafting of non-trivial code, in particular in the case of distributed systems. Consequently, a FCL is often tangled with the source of the target application [1, 2, 27] or composed of several *ad hoc* scripts [5, 17]. As a result, this may lead to inconsistencies and instabilities in the resulting SASS. This is essentially due to the fact that the initial assumptions on the operation conditions for the designed controller are usually not explicitly specified and that no systematic verification is conducted to ensure they still hold at runtime.

In our previous work [25], we have proposed a domain-specific modeling language, called *Feedback Control Definition Language* (FCDL), that is addressing some of the integration challenges related to the visibility of FCLs. It provides system-level abstractions for integrating external control mechanisms into existing software systems, notably through an underlying actor-oriented component model. In this chapter, we go beyond this contribution by focusing on the reliability aspect of the integration activity. In particular, we present an extension of the FCDL language to support a *design-by-contract* methodology [29]. Design-by-contract is a pragmatic and a lightweight method for embedding elements of formal specification into software elements (*e.g.*, objects, components, services). The approach strongly focuses on the correctness of the contracted system and thus contributes to improve the overall system reliability assurance [26]. Concretely, we provide a first-class language support for defining:

- *behavioral contracts* that assert component behavior through state invariants and pre and postconditions,
- *interaction contracts* that express allowed component interactions, and
- *structural and temporal invariants* that define architecture constraints as well as design and execution time interaction invariants.

Next to contracts, we also equip FCDL with first-class support for systematic fault handling. These constructs can be further used to respond to runtime deviations from expected quality of service (*e.g.* response time violation in the sense of a timeout).

Including the support directly into the FCDL language offers several advantages. With reasonable development effort, contracts make the system specification more rigorous and therefore improve its verification capabilities. Static structural and interaction invariants can be checked at design time, allowing developers to verify system consistency and spot design flaws early. The fault handling strategies enable developers to define coherent responses to runtime contract violations, contributing to the construction and maintenance of more fault tolerant systems.

Applied to the engineering of self-adaptive software systems, contracts make the assumptions about the system operation conditions more explicit, and verifiable in a systematic way to handle system failures. Separation of concerns is also promoted as the controller design and integration can be decomposed and implemented by experts in their respective domains and at different levels of abstraction, still under an explicit specification.

The chapter is structured as follows. Section 2 introduces the adaptation scenario we use to illustrate our approach. Section 3 gives an overview of the FCDL language. Section 4 presents the different contract extensions for FCDL while the description of the failure handling support is provided in Sect. 5. The trade-offs of using contracts are discussed in Sect. 6. Section 7 surveys related work and finally, Sect. 8 concludes the chapter.

2 Adaptation Scenario

The adaptation scenario used throughout this chapter is based on the work of Abdelzaher *et al.* [1, 2] on QoS management control of web servers by content delivery adaptation. The reason for using this scenario is that it provides both (i) a control theory-based solution to a well-known and well-scoped problem, and (ii) enough details for its re-engineering. For our illustration, we focus on the single server case with all requests having the same priority.

The aim of the adaptation is to maintain a web server load at a given pre-set value preventing both its underutilization and its overload. The content of the web server is pre-processed and stored in M content trees where each one offers the same content but of a different quality and size (*e.g.* different image quality). For example, let us take two trees /full_content and /degraded_content. At runtime, a given URL request, *e.g.* /photo.jpg, is served from either /full_content/photo.jpg or /degraded_content/photo.jpg depending on the current load of the server. Since the resource utilization is proportional to the size of the content delivered, offering the content from the degraded tree helps reducing the server load when the server is under heavy load.

Figure 1 shows the block diagram of the proposed control. The *Load Monitor* is responsible for quantifying server utilization U . It periodically measures request rate R and delivered bandwidth W . These measurements are then translated into a single value, U . Since service time of a request constitutes of a fixed

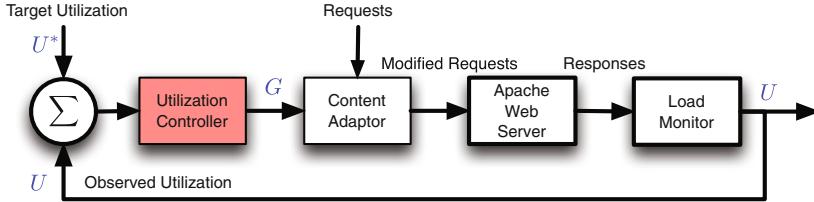


Fig. 1. Block diagram of the adaptation scenario [1]

overhead and a data-size dependent overhead, using some algebraic manipulations, the utilization from the request rate and delivered bandwidth is derived as:

$$U = aR + bW = a \frac{\sum r}{t} + b \frac{\sum w}{t} \quad (1)$$

where a and b are some platform constants derived by server profiling (details in Abdelzaher *et al.* [1]), $\sum r$ and $\sum w$ are the number of requests and the amount of bytes sent over some period of time t , respectively. The *Utilization Controller* is an *integral* (I) controller, which, based on the difference between the target utilization U^* (set by a system administrator) and the observed utilization U , computes an abstract parameter G representing the severity of the adaptation action. This value is used by the *Content Adaptor* to choose which content tree should be used for the URL rewriting. The achieved degradation spectrum ranges from $G = M$, servicing all requests using the highest quality content tree to $G = 0$ in which case all requests are rejected. It is computed as:

$$G = G + kE = G + k(U^* - U) \quad (2)$$

where k is the controller tuning parameter that is determined *a priori* using some control analytic techniques (details in Abdelzaher *et al.* [1]). Shall $G < 0$ then $G = 0$ and similarly shall $G > M$ then $G = M$. If the server is overloaded, ($U > U^*$) the negative error will result in decrease of G which in turn changes the content tree decreasing the server utilization and vice versa.

Next to the integral control, Abdelzaher *et al.* [1,2] also propose a more sophisticated *proportional integral (PI)* controller. For the simplification, in this chapter we only consider the former one, since from the software architecture perspective the only difference between them is the type of AE that is instantiated. The FCDL primary focus is facilitating controller integration not its design, since for this, there already exist sophisticated tools such as MATLAB [20].

This adaptation scenario is essentially a simplified version of the *znn.com case study* [12] in which only the server content fidelity is considered. While in this chapter we are mostly focus on the control theory based solution, there exist other approaches as well. In the related work section, we give an example how FCDL can be used for their integration.

3 Feedback Control Definition Language in a Nutshell

In this section we present the essentials of the FCDL language and illustrate how it can be used to integrate the adaptation scenario from the previous section. We deliberately skip some technical details about the language and instead provide pointers where details can be found. The complete FCDL syntax and semantics can be found in Kříkava's PhD thesis [24] as well as on the FCDL web site¹.

3.1 Modeling Feedback Control Loops

FCDL is a *Domain-Specific Modeling Language* (DSML) tailored for defining feedback control architectures of external self-adaptive software systems—*i.e.*, systems where the adaptation engine is isolated from the target (adaptable) system and interact through identified touchpoints. It is based on an actor-oriented model [22] where the components (actors) are called *Adaptive Elements* (AE). They represent the different FCL processes, such as monitoring, decision making and reconfiguration and feedback control loops are formed by connecting them together.

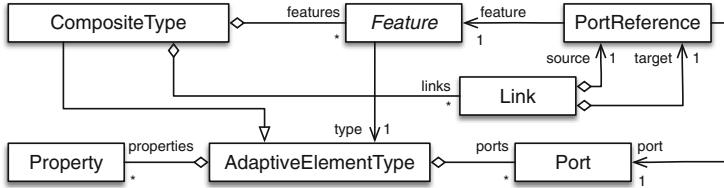


Fig. 2. Excerpt of the FCDL abstract syntax (type meta-model)

Figure 2 shows an excerpt of the FCDL abstract syntax. An AE (`AdaptiveElementType`) has a well-defined interface that abstracts its internal state and behavior. It consists of properties (`Property`), ports (`Port`) and operations². Properties define the AE initial configuration as well as its runtime state variables. The input and output ports are the points of communications through which elements can exchange messages. An operation is a function $input \times state \rightarrow output$ executed upon receiving a message. The concrete function code can be expressed for example in Java.

The execution semantics is based on the Ptolemy [16] push-pull model of computation [45]. Once an AE receives a message, it activates and executes its associated behavior. The result of the execution may or may not be sent further to the connected downstream elements that in turn cause them to active and so on and so forth. An AE can be *passive* or *active*. The former is activated by

¹ <http://fikovnik.github.io/Actress>.

² Not shown in the excerpt, details are in Chap. 3 of Kříkava's PhD thesis [24].

receiving a message while the latter attaches an appropriate event listener to activate itself when an event of interest occurs. Each AE represents a process of a FCL, which may either be:

- a *sensor* collecting raw information about the state of the target system and its environment (*e.g.* log files, hardware sensors),
- an *effector* carrying out changes on the target system using provided management operations (*e.g.* configuration file modification, system calls),
- a *processor* processing and analyzing incoming data both in the monitoring and reconfiguration parts (*e.g.* data filters, converters), or
- a *controller*, a special case of a passive processor that is directly responsible for the decision making (*e.g.* PI controller, rule engine, utility controller).

To enforce data type compatibility, the FCDL modeling language uses static typing. For each port and property one has to explicitly declare the data type that restricts the data values it accepts. To improve reusability, the meta-model additionally supports parametric polymorphism, making adaptive elements work uniformly on a range of data types.

FCDL also supports constructing *composite* components (`CompositeType`) which is also the primary unit of deployment. It defines both the instances of other components (`Feature`) they contain and the connections between the instances ports (`Link`).

Additionally, the language supports distribution and reflection³, thereby enabling coordination and composition of multiple distributed FCLs [24, Sects. 3.3.5 and 3.3.6].

3.2 Illustration

Figure 3 shows how the control mechanism elaborated in the previous section can be integrated in the target system (Apache web server) in FCDL. The figure uses an informal FCDL graphical notation, an informal visual representation of FCDL models. A formal textual syntax is presented in Sect. 3.3 with an example corresponding to this illustration in Listing 1.

The feedback control loop consists of three composites. The `ApacheWebServer` that collects elements related to the Apache web server, `LoadMonitor` that is responsible for computing the server utilization U and `ApacheQOS` that embodies them and assembles the FCL. It works as follows:

- *Decision making.* The controller maps the current system utilization characteristics U into the abstract parameter G controlling which content tree should be used by the web server. In FCDL it is represented by the `utilController` AE that defines one push input port, `input`, for U and one push output port, `output`, for G . Once a new utilization value is pushed

³ Conceptually, each AE can be seen as a target system itself, and as such it can provide sensors and effectors enabling the AE reflection—hence the name *adaptive element*.

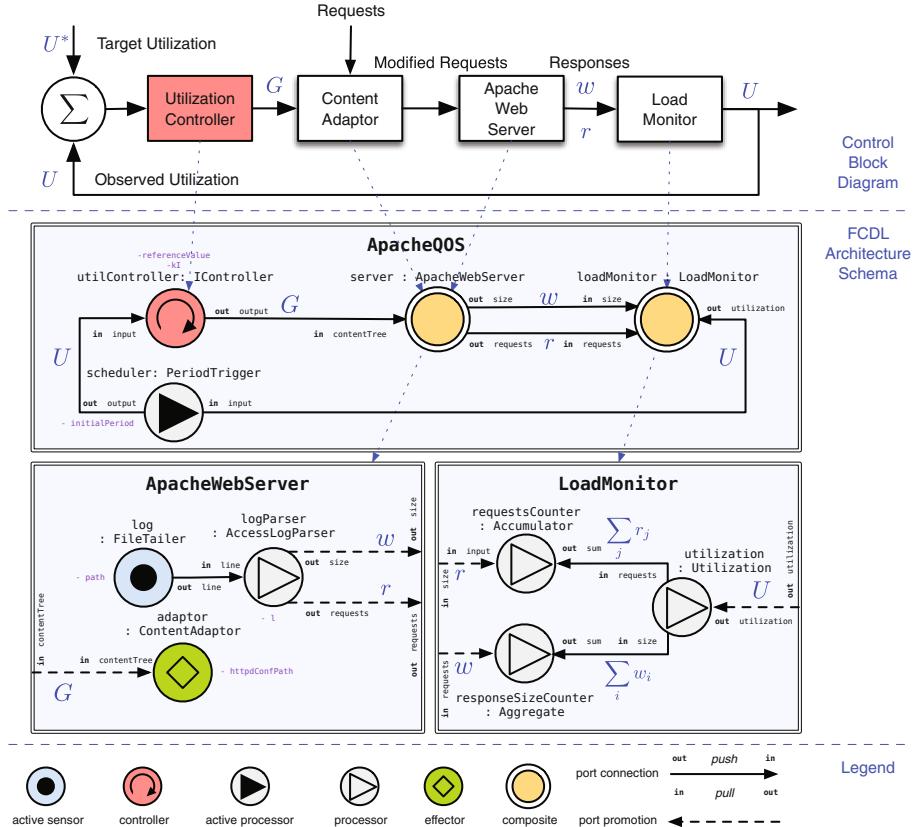


Fig. 3. FCDL schema of the adaptation scenario.

to its input port, it computes G using (2) and pushes the result to the output port. The **IController** also includes properties for the controller set point **referenceValue** and the K_I controller integral gain kI .

- *Monitoring.* The system utilization U depends on request rate R and bandwidth W . Both information can be obtained from Apache access log file. We create an active sensor, **FileTailer**, that activates every time a content of a file changes and sends the new lines over its push output port. It is connected to **logParser** that parses the incoming lines and computes the number of requests r and the size of the responses w , pushing the values to the corresponding **requests** and **size** ports. Consequently this increments the values of two connected counters **requestCounter** and **responseSizeCounter**, implemented as simple passive processors that accumulate the sum of all received values.

To compute utilization U , the sum of requests $\sum r$ and response size $\sum w$ has to be converted to request rate R and bandwidth W —*i.e.*, the number of requests and bytes sent over a certain time period t . One way of doing this

is by adding a `scheduler`, an active processor that every t milliseconds pulls data from its pull input port (the pull is indicated by the arrow going from the `scheduler` input port to the `utilization` output port) and in turn pushes the received value to its output port. Essentially, it is a scheduler that acts as a mediator between the two connected AEs. In this scenario, it is responsible for the timing of the FCL execution. By pulling data from its input port, it activates the `utilization` processor that (i) fetches the corresponding sums of requests $\sum r$ and response sizes $\sum w$ using the two pull input ports; (ii) converts them to request rate R and bandwidth W ; and (iii) finally computes U using (1). The resulting utilization is then forwarded by the scheduler into the `utilController`.

- *Reconfiguration.* Upon receiving the extent of adaptation G , the Content-Adaptor reconfigures the web server URL rewrite rules so that the newly computed content tree is used to serve the upcoming requests.

3.3 Modeling Support

The idea behind a DSML, such as FCDL, is to raise the level of abstraction, which should lower accidental complexities and in turn increase productivity. For this to be true, however, DSMLs have to be associated with software development tools that automate tasks such as model construction and code generation [42].

The model creation is facilitated by a textual *domain-specific language* (DSL). It is built using the Xtext⁴ software language engineering framework. The Listing 1 shows a code excerpt of the adaptation scenario (cf. Fig. 3).

```

active processor PeriodicTrigger<T> { // polymorphic processor with T parameter
    push in port output: T
    pull in port input: T
    self port selfport: Long // active AE contains a self port for self-activation
    property initialPeriod: Long
    // ...
}

controller IController {
    push in port input: Double
    push out port output: Double
    // ...
}

composite ApacheQOS {
    // instantiate a contained AE with a concrete data type for the T parameter
    feature scheduler = new PeriodicTrigger<Double> {
        initialPeriod = 5.seconds // set properties
    }
    feature utilController = new IController { /* ... */ }
    // port connection
    connect scheduler.output to utilController.input
    // ...
}

```

Listing 1. Adaptation scenario FCDL code excerpt

FCDL model is used as an input to a model transformation that synthesizes an executable system based on Akka⁵, an actor-based framework and runtime

⁴ <http://eclipse.org/Xtext>.

⁵ <http://akka.io>.

for the Java virtual machine. Because the FCDL model is already an actor-oriented model, the source code transformation is rather straightforward. Essentially, each AE type is turned into a Java class with generated skeleton methods left for developers to complete with the logic (*e.g.* parsing access log records in the `ApacheLogParser`). These classes are used as delegates translating the lower level actor interactions (*e.g.* messages initializing actors) into corresponding method calls (*e.g.* initialization or activation method). Using this pattern, developers never have to deal with any lower-level actor API making it portable across different actor frameworks. This also simplifies AE testing, which can be done in isolation without any actor runtime.

4 Adaptive Element Contracts

We now present a set of extensions to FCDL for adaptive element contracts specification. The aim is to make the assumptions about the operation conditions explicit. In FCDL, this can be realized at the architecture level by specifying invariants that constrain adaptive element structure, interactions, and behavior.

We start by introducing *Interaction Contracts* (IC) as they define the semantics of an adaptive element (AE) execution. Next, we extend ICs with *Behavioral Contracts* (BC) that assert AE behavior. Finally, we complete the section with adaptive element structural and temporal invariants.

In this section, we only focus on the contract definition—*i.e.*, on the definition of the *expected* behavior. The next section will discuss when contract violation and will detail the mechanisms to handle *exceptional cases*.

4.1 Interaction Contracts

Motivation. Let us consider a more sophisticated version⁶ of the Accumulator from our adaptation scenario (*cf.* Fig. 4). It works as follows: When it receives data on its input port, it pushes to its output port the received value plus the sum of all the input values it has received since the last time the `reset` port was triggered. Similarly, when pulled on the `sum` port, it returns the sum of all the input values since the last reset. Finally, when any data are pushed to its `reset` port, it sets the accumulated value back to 0.

```
processor Accumulator {
    in push port input: Long
    in push port reset: Boolean
    out pull port sum: Long
    out push port output: Long
}
```

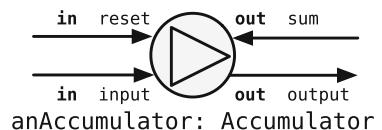


Fig. 4. Improved Accumulator processor

⁶ Inspired by the Ptolemy 2 Accumulator actor cf. <http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII8.1/ptII/doc/codeDoc/ptolemy/actor/lib/Accumulator.html>.

The above description makes the element interactions rather intuitive. However, the fact that every time an input is received data will be pushed over the output port is not explicitly stated in the architecture. It is but only mentioned in its documentation. Such architecture underspecification leads to several shortcomings:

1. There is no systematic way to verify that an implementation matches its documentation.
2. In more complex cases or with less rigid documentation, it may result in different interpretations and thus incompatible implementations.
3. The implementation has to manually follow the data flow to check which AE *functionality* should be executed, which is tedious in cases when multiple inputs have to be synchronized.
4. The element interactions are an integral part of its implementation, strongly reducing the possibility of formal analysis.

To address this architecture underspecification issue, we enrich the adaptive element definition with *interaction contracts* that express allowed interactions, making them explicit.

In the following we give an overview of the interaction contracts that have been implemented in the FCDL language based on the work of Cassou *et al.* [9] (cf. Sect. 7). The complete formal specification of interaction contracts is available in Kříkava's PhD thesis [24, Sect. 4.2 from page 74].

Contract Specification. The objective of an interaction contract is to describe the interactions allowed by an adaptive element. More formally, we define a *basic interaction contract* as a tuple $\langle A; R; E \rangle$ where

- A is the activation condition that indicates what interactions activate the AE—*i.e.*, a set of push (\uparrow) input ports or a pull (\downarrow) output port;
- R is the data requirements denoting what additional data might be needed during its execution—*i.e.*, a set of pull input ports—and
- E is the data emission—*i.e.*, a set of push output ports through which the results of computation will be distributed.

For example, the interaction contract associated with `PeriodicTrigger` is $\langle self; \downarrow(\text{input}); \uparrow(\text{output?}) \rangle$ denoting an interaction caused by *self* activation (as it is an active processor) where *input* port might be pulled and conditionally (?) data pushed to the *output* port. In FCDL this corresponds to the following definition:

```
active processor PeriodicTrigger<T> {
    push out port output: T
    pull in port input: T
    port selfport: Long

    act activate(selfport; input; output?) // <self;↓(input);↑(output?)>
}
```

For adaptive elements with multiple operations, an interaction contract is a composition (\parallel) of all the individual ICs. For example, the IC of the Accumulator described above is a composition of three basic ICs $\langle \uparrow(\text{input}); \emptyset; \uparrow(\text{output}) \rangle \parallel \langle \downarrow(\text{sum}); \emptyset; \emptyset \rangle \parallel \langle \uparrow(\text{reset}); \emptyset; \emptyset \rangle$ which in FCDL corresponds to:

```
processor Accumulator {
    in push port input: Long
    in push port reset: Object // any data pushed will reset the counter
    out pull port sum: Long
    out push port output: Long

    act onInput(input; ; output) //  $\langle \uparrow(\text{input}); \emptyset; \uparrow(\text{output}) \rangle$ 
    act onSum(sum; ; ) //  $\langle \downarrow(\text{sum}); \emptyset; \emptyset \rangle$ 
    act onReset(reset; ; ) //  $\langle \uparrow(\text{reset}); \emptyset; \emptyset \rangle$ 

    // ...
}
```

In the case of composites, an IC is automatically inferred [24, Sect. 4.2.4 on page 84] based on the ICs of the contained AEs. For example, the ApacheWebServer IC is specified as $\langle \text{self}; \emptyset; \uparrow(\text{requests}, \text{size}) \rangle \parallel \langle \uparrow(\text{contentTree}); \emptyset; \emptyset \rangle$.

Denotation. A denotation of an IC $\langle A; R; E \rangle$ is a function $A \times R \rightarrow E$. Each of the contracts can then be used to synthesize an AE activation method. For instance, the following listing shows the generated Java interface that corresponds to the Accumulator:

```
public interface Accumulator {
    Long onInput(Long input); //  $\langle \uparrow(\text{input}); \emptyset; \uparrow(\text{output}) \rangle$ 
    Long onSum(); //  $\langle \downarrow(\text{sum}); \emptyset; \emptyset \rangle$ 
    void onReset(Object reset); //  $\langle \uparrow(\text{reset}); \emptyset; \emptyset \rangle$ 
}
```

Since the information about AE activation are explicitly stated, the low-level details about input data synchronization can be automatically generated. Developers therefore only need to focus on the actual functionality. The synthesized methods further help the implementation in the way that they are both *prescriptive*—*i.e.* guiding the developer—and *restrictive*—*i.e.* limiting the developer to what the architecture allows—in contrast to having only one monolithic activation method.

Input Synchronization and Disjunction. Interaction contracts support complex activation patterns, such as input synchronization and input disjunction. Let us consider a passive adaptive element \mathcal{A} with two input push ports (in_1, in_2) and one push output port (out). An interaction contract $\langle \uparrow(\text{in}_1, \text{in}_2); \emptyset; \uparrow(\text{out}) \rangle$ synchronizes the input ports and will only activate \mathcal{A} if data have been pushed to both input ports. This corresponds to a Java method:

```
<T> T onIn1In2(T in1, T in2); //  $\langle \uparrow(\text{in}_1, \text{in}_2); \emptyset; \uparrow(\text{out}) \rangle$ 
```

On the other hand a input disjunction (\vee) used in the activation condition of an IC such as $\langle \uparrow (\text{in}_1 \vee \text{in}_2); \emptyset; \uparrow (\text{out}) \rangle$ will activate \mathcal{A} any time any of the input ports receives data. In Java:

```
<T> T onIn1In2(T in); // ⟨↑ (in1 ∨ in2); ∅; ↑ (out)⟩
```

Finally, an IC $\langle \uparrow (\text{in}_1); \emptyset; \uparrow (\text{out}) \rangle \parallel \langle \uparrow (\text{in}_2); \emptyset; \uparrow (\text{out}) \rangle$ will also activate \mathcal{A} any time there has been a data pushed on any of its input ports. However, the crucial difference here is that, in this latter case, there is a different behavior associated with each of the basic contracts corresponding to two different activation methods:

```
<T> T onIn1(T in1); // ⟨↑ (in1); ∅; ↑ (out)⟩
<T> T onIn2(T in2); // ⟨↑ (in2); ∅; ↑ (out)⟩
```

Architecture Properties. Interaction contracts make the component interactions explicit. The resulting architecture is therefore amenable to automated analysis and the following properties can be statically checked at design time [24, Sects. 4.2.5–4.2.7].

- *Consistency.* ICs not only define the elements interactions but also imply certain interactions requirements for the other elements in order for the assembly to be consistent. For example, for the scheduler (cf. Fig. 3) to be able to pull data from its input port, it must define a contract with the following data requirement: $R = \Downarrow (\text{input})$. This implies that the connected element, loadMonitor, must have in one of its ICs and activation condition $A = \Downarrow (\text{output})$ since output port is connected to the PeriodicTrigger input port.
- *Determinacy.* An interaction contract can be composed of one or more basic interaction contracts defining multiple activation conditions for an AE. It is important to make sure that these activation conditions do not interfere with each other. An interference occurs for example if two or more interaction contracts share the same push input port in their activation condition. In such a case, the activation is not *deterministic* since it is not possible to identify which contract should be executed when the input data arrives.
- *Completeness.* An adaptive element might define a multitude of interaction contracts from which some are required and some are optional. For example, in the case of the Accumulator, the interaction contract linked to the reset functionality is optional while the other two are required. It is therefore not possible to use the element without connecting its input and sum ports while the reset might be left out.

4.2 Behavioral Contracts

Motivation. The interaction contracts precisely specify which adaptive element behavior is triggered by what interaction. However, they say very little

about the behavior itself. This might be problematic from at least two reasons. First, from the correctness point of view, less constrained implementation may lead to a higher chance of containing bugs than an implementation satisfying well-understood specifications [29]. Second, from the robustness perspective, Cámaras *et al.* [8] showed that the unconstrained inputs and outputs may lead to salient failures, which are both hard to detect and may cause unexpected or undesired behavior. In the case of self-adaptive software systems, this is particularly worrying as the controller may steer the system into a wrong state without any obvious reason.

A systematic approach to address both concerns is to use behavioral contracts—*i.e.* pre and postconditions and state invariants. They have been successfully used in other programming languages, such as Eiffel⁷, OCL [33] or Scala [34]. In our case, these contracts will allow developers to augment the type specification and express AE properties and behavioral requirements.

State Invariants. A state invariant asserts the values of AE properties since that is the only component of the adaptive element definition that is directly modifiable by users (during AE instantiation in a composite). It is specified as a boolean expression using a `state inv` construct.

Let us consider the `AccessLogParser` from the adaptation scenario (cf. Fig. 3), which is responsible for parsing the Apache access log file. To accommodate for different log formats, we define a `logFormat` property:

```
property logFormat: String
```

However, we need to make sure that it is always set and that the value is a valid regular expression. In FCDL, these concerns are expressed using state invariants. First, we define an invariant that makes sure the property is set to a non-empty string:

```
state inv NonEmptyLogFormat = self.logFormat != null && self.logFormat.length > 0
```

Second, we ensure a valid regular expression:

```
1 state inv ValidLogFormat if NonEmptyLogFormat = new StateInvariant {
2     override check() {
3         try {
4             java.util.regex.Pattern::compile(logFormat)
5             pass() // an invariant is satisfied
6         } catch (java.util.regex.PatternSyntaxException e) {
7             fail("Invalid pattern: "+e.message) // invariant is violated
8         }
9     }
10 }
```

The second invariant shows an alternative implementation that uses an anonymous class implementing a designated interface. It also shows some additional features that are supported by the FCDL invariants. On line 1 we define

⁷ First appeared in the Eiffel language under the name *Design-by-contract* [30].

a dependency between invariants—*i.e.*, the check will only be evaluated if `NonEmptyLogFormat` has not been violated (that is why we can skip a nullity check for `logFormat`). On line 7 we further provide a user-friendly error message with more details about what went wrong.

Assertion Language. The code listing above reports examples of typical assertions used in BCs. Assertions are boolean expressions that come from the Xbase language [15]. Xbase is a Java-like expression language especially designed to be embedded in DSLs that are created using the Xtext language engineering workbench. It is interoperable with Java and the expressions can instantiate Java classes, implement Java interfaces and call Java methods. The language is statically typed and it uses type inference to provide type safety without unnecessary syntactic clutter. It includes support for first-order logic collection operations, which makes assertions, such as $\forall x \in T.p(x)$ or $\exists x \in T.p(x)$, convenient to define using expressions like `T.forall[x | p(x)]` and `T.exists[x | p(x)]`.

Many of the assertions are usually simple expressions for which Xbase provides a suitable option. However, it might not always be the case and complex assertions can be equally implemented in Java. A developer can either instantiate an existing class that conforms to the right interface or omit the expression in which case a skeleton Java class will be generated instead. This gives a possibility to reuse state invariants across adaptive elements.

Next to the BCs, Xbase can also be used to directly implement AE operations in FCDL.

Pre/Postconditions. Pre and postconditions are related to the executions of AEs operations. They are specified as boolean expressions using the `require` and `ensure` keywords.

Let us consider again the `AccessLogParser`. To prevent potential salient failures, we should check that the received input line matches the specified access log format. In FCDL, we can express it using the following precondition:

```
act activate(line; ; requests, size) { // ⟨↑(line); ∅; ↑(requests, size)⟩
    require LineMatchingPattern if LineNotEmpty = new PreCondition {
        val Pattern = java.util.regex.Pattern::compile(self.logFormat)
        override check() {
            val m = Pattern.matcher(line)
            assertTrue("The input line must match the log format", m.matches())
        }
    }
}
```

The structure is similar to the structural invariant definition. The main difference is that, in its scope, it can additionally access input data (`line`), which is injected automatically into the class definition. Like Java anonymous classes, it also allows one to declare variables and constants (`Pattern`). We can therefore express more complex invariants—*e.g.* to ensure that the log entries are successive:

```

require SuccessiveTimestamp if ValidTimestamp = new PreCondition {
    val Pattern = java.util.regex.Pattern::compile(self.logFormat)
    val TimestampFormat = new SimpleDateFormat("dd/MMM/YYYY:HH:mm:ss Z")
    var lastTime = new Date(0)

    override check() {
        val timestamp = Pattern.matcher(line).group("time")
        val time = TimestampFormat.parse(timestamp)

        if (time.after(lastTime)) {
            lastTime = time
            pass()
        } else {
            fail("Invalid record: "+time+" appears before the last record")
        }
    }
}

```

It is important to note that it makes perfect sense to express this as a contract instead of having the same check within the operation method of the adaptive element. The reason is that non-successive timestamps present a deviation from the expected behavior, concretely a bug in the connected component that supplies the invalid log entries (log in this case), and not a special case that should be handled in the operation body. Following this approach we clearly express the assumptions on the adaptive element inputs.

4.3 Interaction Invariants

Motivation. So far, we have presented contracts that consider adaptive elements in isolation. While it is important to express assumptions on the interactions and behavioral properties of individual components, these contracts do not provide any assumptions about the complete loop architecture—*i.e.* about the feedback control they implement. These properties mostly include *liveness* (*e.g.* data collected by a given sensor always trigger a particular reconfiguration action) and *safety* (*e.g.* data from a given sensor never lead to a certain reconfiguration). They are usually expressed in temporal logic [38]. In FCDL, we use *Linear Temporal Logic* (LTL) to characterize these properties in the sense of interaction invariants.

Static Interaction Invariants. An interaction contract constrains the interactions allowed at the level of a single adaptive element. An interaction invariant does the same but at the composite level. It expresses requirements and restrictions about the data flow within an assembly of components.

For example, in the adaptation scenario we would like to ensure that whenever someone accesses the web server a controller will be activated. Using the LTL, this invariant can be expressed as:

$$\square (\text{log_activate} \rightarrow (\Diamond \text{utilController_activate}))$$

The predicate variables `log_activate` and `utilController_activate` relate to the `log` and `utilController` elements and are `true` when they have been activated. The LTL formula means that: “*always* (\square) when the `log activate`

interaction contract is executed, then the utilController activate interaction contract will *eventually* (\lozenge) be activated as well.”

In FCDL this interaction invariant is defined in the ApacheQOS composite using the following code (both the FileTailer and IController define one interaction contract called activate):

```

composite ApacheWebServer {
    feature log = new FileTailer /* ... */
    // ...
}

composite ApacheQOS {
    feature server = new ApacheWebServer /* ... */
    feature utilController = new IController /* ... */
    // ...

    temp inv loopLiveness =
        // LTL formula
        [] (activate@server.log -> <> activate@utilController)
}
}

```

These properties can be easily verified by an appropriate model checker, such as SPIN [23]. Because the verification is done statically at design time, we refer to these interaction contracts as *static*. SPIN takes a model of a system described in the Promela modeling language. It will try to find a counter example in which the negated LTL formula holds providing the corresponding stack trace. The Promela model can be generated from our architecture model by mapping the element interaction contracts and message flow into the corresponding Promela concepts. For example, the utilController : IController interaction contract $\langle\uparrow(\text{input}); \emptyset; \uparrow(\text{output})\rangle$ is translated into Promela code shown in Listing 2.

```

// act activate(input; ; output)
#define utilController_activate (util_controller@act_activate)

// ports
chan utilController_port_input = [1] of { mtype }; // push in port input: Double
chan utilController_port_output = [1] of { mtype }; // push out port output: Double

active proctype utilController() {
    byte act = 0;

    end: // infinite process

    waiting: // waiting for activation
    if
    :: utilController_port_input ? PUSH -> act = 1; // act activate(input; ; output)
    fi;

    executing: // element activations
    if
    :: act == 1 -> // act activate(input; ; output)
        act_activate:
        utilController_port_output ! PUSH;
    fi;

    act = 0;
    goto waiting;
}

```

Listing 2. Example of the generated Promela code for IController

Since the interaction contracts are also used to synthesize the AE operation methods, these properties are also preserved at the implementation level.

Timed Interaction Invariants. The static interaction invariants above allowed us to define an invariant ensuring that, whenever someone accesses the web server, a controller will be activated. The natural extension is to add a time constraint—*i.e.*, to ensure that not only something will happen, but also to put a deadline until when it has to happen. We refer to these types of constraints as *timed interaction invariants*. They extend the static interaction invariants with quantitative time.

The following formula adds a 6 s (5 s is the initial scheduling period –cf. Listing 1) deadline for the controller activation from the new access log record:

$$\square(\text{log-activate} \rightarrow (\Diamond_{\text{in} < 6\text{seconds}} \text{utilController.activate}))$$

In FCDL it is expressed as:

```
temp inv loopLivenessWithDeadline =
[] (activate@server.log -> <> (in < 6.seconds) activate@server.adaptor)
```

Similarly, we can track a progress of an individual component. For example, we might want to ensure that the controller is triggered regardless the web server activity every at least 6 s:

```
temp inv controllerLiveness =
[] (<> (in < 6.seconds) activate@utilController)
```

Unlike the static interaction invariants, the timed invariants are only verified at runtime. Our realization is based on the approach developed by Stoltz and Bodden [43] for temporal assertions of Java-based programs.

4.4 Structural Invariants

Motivation. Interaction contracts make possible to check architecture consistency. However, because of their generality, they do not help to spot domain-specific problems and FCL architecture issues. For example, an effector that is manipulated by multiple controllers may lead to undesired interference [19].

Essentially, these problems are related to the model structure, concretely the structural configuration of the adaptive elements forming the FCLs. A general mechanism to constraint model structure is provided in languages like OCL that defines model invariants as boolean assertions over model elements structure [33]. In a similar way, we extend the FCDL language with structural invariants that operate on the FCDL meta-model and make possible to statically assert adaptive element structures at design time.

Invariant Specification. A structural invariant operates at the level of the FCDL instance meta-model (cf. Fig. 5), which corresponds to the type meta-model shown in Fig. 2. For example, instead of using an instance of `PeriodicTrigger`, which is an instance of `AdaptiveElementType`, we work with the

instance of `scheduler`, which is an instance of `AdaptiveElementInstance` (that has a reference called `type` pointing to the `AdaptiveElementType`, concretely to `PeriodicTrigger`).

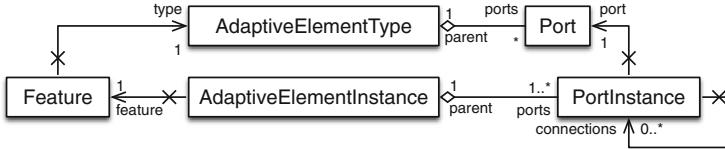


Fig. 5. Excerpt of the FCDL instance meta-model

For example, to ensure that the `requests` and `size` ports of the Utilization processor are connected to two different sources (in order to prevent mistakenly connecting them to the same Accumulator for instance), we use the following code that traverses the FCDL instance model (`self` is an instance of the currently checked `AdaptiveElementInstance`):

```

struct inv DifferentInputSources =
  self.ports
    .filter[p | p.name = 'size' || p.name = 'requests'] // select ports
    .map[p | p.connections] // select their connects
    .map[p | p.parent] // select owning instances
    .toSet
    .size == 2 // there must be two different ones
  
```

Similarly to the other contracts introduced in this section, structural invariants also support dependencies (to other structural constructs) and can be defined using Xbase anonymous classes or Java implementations.

5 Failure Handling

The contracts introduced in the previous section aim at explicitly specifying architecture-level and behavioral-level assumptions about feedback control loops—*i.e.* they define what is *expected*. This section focuses on the opposite *exceptional* cases—*i.e.* on what happens when these contracts are violated. It describes the mechanisms provided in FCDL that can be used to detect exceptions and to coherently handle them.

5.1 Failures and Exceptions

When associated to contracts, Meyer [30] defines an *exception* as a runtime event that may cause a *failure*—*i.e.* the termination of a routine call that does not satisfy the routine contract. What is important to note on this definition is that every failure is the result of an exception, whereas not every exception leads to a failure. An exception occurs when an operation cannot achieve its intention. In FCDL there are two sources of exceptions: a *contract violation* of either an

invariant (state or timed temporal) or a pre/postcondition, or a *runtime exception* that has been thrown by an adaptive element operation method⁸. Possible causes of runtime exceptions fall into one of the three categories: *systematic* caused by programming errors, *accidental* caused by corrupt internal state, or *transient* caused by failures of some external resource used during computation including exceptions caused by connected elements (e.g. timeouts).

As previously stated, an exception does not necessarily need to lead to a failure and it is possible to detect and recover from an exceptional state. In FCDL, actors are organized in composites. This introduces a hierarchical structure (cf. Fig. 6) whereby a composite supervises its subordinates (nested adaptive elements). This therefore implies a dependency relation between the actors. Besides that, the composite is responsible for routing messages from promoted ports to the connected adaptive elements and vice-versa, it must also respond to their failures. A subordinate failure then becomes an exception in a supervisor.

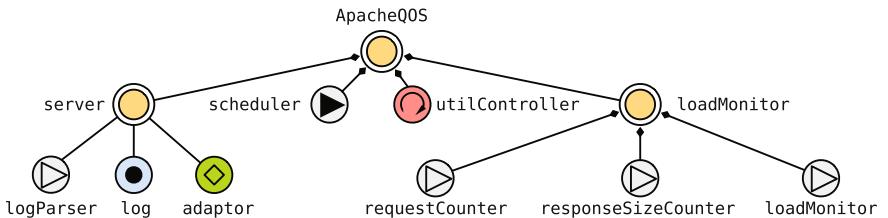


Fig. 6. Adaptive Elements hierarchy

Depending on the nature of the exception, the supervisor can choose one of the following actions⁹: *resume* and thus keeping the state of the error adaptive element as it is, *restart* adaptive element(s) with a potentially different configuration or implementation, or finally *escalate*. An escalation converts the exception into a failure, which in turn, either becomes an exception in a higher-level composite or fail the complete system when there are no more layers.

5.2 Exception Handling

The goal of exception handling is to make coherent responses to exceptional cases. These are exceptions that are not handled within AE operation methods themselves, since otherwise they would be expected regardless any special treatment they require. Exceptions are thus handled at the composite level.

⁸ By the term *runtime exception*, we mean all exceptions that are possibly thrown at runtime, which in the case of Xbase and Java include both checked and unchecked exceptions.

⁹ Making a parallel with the Meyer's *Disciplined Exception Handling principle* [30], the resume and restart actions corresponds to the *retrying* response and the escalation falls into *failure*.

The following listing shows the exception handling constructs:

```
composite C {
    // ...
    catch {
        case [<variable_name>:] <exception_sel> @<feature_sel> = ...
        // ...
    }
}
```

<exception_sel>@<feature_sel> is an expression that specifies which exceptional case(s) are to be handled—*i.e.* which exception(s) from which adaptive element(s). It has two parts: a set of exception types (or an asterisk matching them all), and a set of considered features in which the exception can occur (or an asterisk matching all features declared within the composite). For example **case** ValidLogFormatViolation @ logParser handles the case of ValidLogFormat contract violation that occurs in logParser instance of AccessLogParser adaptive element.

Before we show how the above constructs can be used to handle an exception, we need to discuss where are the exceptions defined. As we have hinted, there is a tight relation between a contract violation and an exception. Concretely, all state and timed interaction invariants as well as both pre and postconditions are turned into exceptions (whose name matches the contract name with the “Violation” suffix¹⁰). Turning contract violations into exceptions allows us to respond to all abnormal cases in a consistent way. As mentioned above, the two possible sources of exception are contract violations, which we have just turned into exceptions, and runtime exceptions, which are all the exceptions possibly thrown by the operation methods.

To illustrate the ramification of unified exceptions, let us consider the AccessLogParser again. Its IC is **act** activate(line; ; requests, size) which, together with the two preconditions defined in the previous section, translates into the following Java interface:

```
public interface AccessLogParser {
    Tuple2<Int, Long> activate(String line)
    throws LineMatchingPatternViolation, SuccessiveTimestampViolation;
}
```

Now, if we had modified the interaction contract to:

```
act activate(line; ; requests, size) throws NumberFormatException
```

the NumberFormatException would have been added to the method throws clause, allowing us to handle¹¹ it in the supervisor using:

```
case NumberFormatException @ logParser
```

¹⁰ There are two reasons for the suffix: (i) it makes a clear distinction between contract logic and contract exception, and (i) it makes more sense English wise.

¹¹ It is unlikely that such an exception would occur. However since the size of the response has to be converted from a String into a Long declaring this runtime exception explicitly contributes to the AE robustness.

By making both exception sources explicit, we can statically check at design time whether all cases are covered.

5.3 Supervision Strategies

Equipped with the exception handling constructs, coherent strategies can now be defined to respond to exceptional cases. Essentially, a strategy is a function that, given an exception context—*i.e.* the exception itself and the reference to the failed adaptive element instance—executes one or more actions. Similarly to contract definition, it can be a Xbase expression including an anonymous class or an existing class instance, or can be left empty so that an empty handler class is generated. The implementation should result into one of the three possible actions. The simplest case is to resume the operation leaving the state of the failed adaptive element untouched. This corresponds to an empty implementation. The second option is to restart one or more adaptive elements depending on how directly the elements affect one another. One can either create an *one-for-one* strategy in which only the failed element is restarted or an *all-for-one* where all inextricably linked elements are restarted. By restarting an element, not only its internal state is cleared, but it also gives an opportunity to select a different configuration—*i.e.* combination of implementation property settings. Finally, by re-throwing the exception the issue is escalated to the higher level of the hierarchy.

Often a combination of actions can be used to define an exception handling strategy. In the following example, we handle a timeout exception that might occur in the PeriodicTrigger when pulling data from a sensor (`aSensor`) through the input pull port. This shows an example of a quality-of-service violation in which the contracted part, the sensor, does not meet its response time:

```

catch {
    case e: InputPortTimeoutViolation@scheduler = new ExceptionHandler {
        var failures = 0
        override handle() {
            failures += 1
            switch failures {
                case failures < 5 : {} // no action = resume action
                case failures > 5 && failures < 10 :
                    switch aSensor { // restart action
                        case SensorImpl1 : restart(aSensor, new SensorImpl2)
                        case SensorImpl2 : restart(aSensor, new SensorImpl1)
                    }
                default : throw e // escalation
            }
        }
    }
}

```

The realized strategy¹² alternates between two different sensor implementations up to a point where it gives up and escalates the problem.

¹² The implementation is rather naive, as the purpose is to demonstrate the language features.

6 Assessment

FCDL models an architecture of a feedback control system. In general architecture models are used for two main engineering concerns: for statical analysis and for mapping the architecture into an implementation [35]. A key element in both cases concerns the amount of details required about components and their interactions in the architectural description which determines the degree of available implementation guidance and verification support. The use of contracts in FCDL increases the amount of details with explicit assumptions about the adaptive element interactions, behavior and structure. This in turn contributes to a better programming and verification support. However, as there is no silver bullet [7], contracts come with their own costs and drawbacks. In this section we assess their impact and trade-offs.

6.1 Modeling with Contracts

For the adaptation scenario from Sect. 2, we have implemented in total 45 contracts (7 interaction contracts, 13 state invariants, 8 preconditions, 10 post-conditions, 1 structural invariant, 1 static and 1 timed interaction invariant). The summary of all these contracts is presented in Table 1. While the control presented in the chapter is of a rather modest size, it represents a real-world adaptation scenario and gives us an interesting insights into modeling feedback control loops with contract support. It allows us to do a preliminary assessment on the impact of the contracts on assumptions *visibility*, implementation *effort*, *performance*, and the overall system *reliability*.

Visibility. One of the design goal of FCDL is visibility—*i.e.* FCLs, their processes and interactions should be made explicit at design time as well as at runtime. The contracts contribute to this goal by making visible not only the interaction but also the assumptions about their behavior at the level of a single element as well as at the level of the assembly.

The explicit specification also helps the separation of concerns between control engineers and software developers. Control engineers can use FCDL to define the overall FCL architecture specifying required assumptions on the loop components whose implementation can be then carried out by developers. Contracts therefore helps to mediate communication and improve adaptive element documentation.

Implementation Effort. The FCDL contracts implement what has been acknowledged to be a reasonable trade-off between a full extend of formal specifications and acceptable effort to developers [31]. Without contracts, the only option would be to follow defensive programming to check and protect incorrect input and invalid state using control flow constructs. Contracts on the other hand make these checks explicit and separate them from the operational method code. Furthermore, they allow developers to systematically handle exceptional cases.

Table 1. Interactions contracts for the adaptation scenario (cf. Sect. 2).

Adaptive element	Contract type	Definition
FileTailor	Interaction	$\langle self; \emptyset; \uparrow (\text{line}) \rangle$
	State	<code>path</code> is non empty
	State	<code>path</code> exists
	State	<code>path</code> is a file
	State	<code>path</code> is readable
	Post	<code>line</code> is not empty
AccessLogParser	Interaction	$\langle \uparrow (\text{line}); \emptyset; \uparrow (\text{requests}, \text{size}) \rangle$
	State	<code>logFormat</code> is not empty
	State	<code>logFormat</code> is a valid pattern
	State	<code>logFormat</code> includes a named group for timestamp
	State	<code>logFormat</code> includes a named group for response size
	Pre	<code>line</code> matches the <code>logFormat</code> pattern
	Pre	<code>line</code> contains a valid timestamp
	Pre	successive log timestamps
	Runtime	reported response size is a valid integer
	Post	<code>size > 0</code>
Accumulator	Interaction	$\langle \uparrow (\text{input}); \emptyset; \uparrow (\text{output}) \rangle \parallel \langle \downarrow (\text{sum}); \emptyset; \emptyset \rangle \parallel \langle \uparrow (\text{reset}); \emptyset; \emptyset \rangle$
	Pre	<code>input</code> \leq <code>Long.MAX_VALUE</code> – <code>value</code>
	Pre	<code>input</code> \geq <code>Long.MIN_VALUE</code> – <code>value</code>
	Post	<code>value == old.value + input</code>
	Post	<code>output == value</code>
	Post	<code>value == 0</code>
LoadMonitor	Interaction	$\langle \downarrow (\text{utilization}); \downarrow (\text{requests}, \text{size}); \emptyset \rangle$
	State	<code>a</code> is set in range cf. (1)
	State	<code>b</code> is set in range cf. (1)
	Pre	<code>requests >= 0</code>
	Pre	<code>size >= 0</code>
	Post	utilization is computed using (1)
	Structural	<code>requests</code> and <code>size</code> are connected to different targets
	Runtime	<code>requests</code> pull input port timeout
	Runtime	<code>size</code> pulls the input port timeout
PeriodicTrigger	Interaction	$\langle self; \downarrow (\text{input}); \uparrow (\text{output}?) \rangle$
	State	<code>initialPeriod > 0</code>
	Post	<code>output == input</code>
	Runtime	<code>input</code> pulls the input port timeout
IController	Interaction	$\langle \uparrow (\text{input}); \emptyset; \uparrow (\text{output}) \rangle$
	State	<code>referenceValue > 0</code>
	State	<code>kI > 0</code>
	Post	<code>output</code> is computed using (2)
ContentAdaptor	Interaction	$\langle \uparrow (\text{contentTree}); \emptyset; \emptyset \rangle$
	Pre	$0 \leq \text{contentTree} \leq M$
ApacheQOS	Temp Int. Inv.	$\square (\Diamond (\text{in} \leq 32s) \text{utilController_activate})$
	Static Int. Inv.	$\square (\text{server.log_activate} \rightarrow \Diamond \text{server.adaptor_activate})$

Implementing all the contracts from Table 1 is associated with some development effort. In the case of behavioral contracts, it was slightly higher than having similar checks directly intertwine in the AE operational methods. Also, as the number of contracts increases the complexity of handling them rises as well. On the other hand, making the exceptions explicit allows one to statically check whether all the cases are covered which would have to be otherwise done manually.

Performance Impact. As with any runtime verification, there is a certain overhead in instrumenting software systems. In the case of FCDL contracts, this includes the penalty of evaluating the assertions, as well as the cost of the hooks that trigger them. In the case of the behavioral contracts, the impact is small since the actual check is synthesized into corresponding AE operation methods. In the case of timed interaction invariant, the instrumentation comes at a cost of an extra actor per invariant and extra notification messages. This overhead is linear to the number of formulae and to the number of predicates they contain.

In the current Akka 2.0 based prototype, the memory overhead is about 400 bytes per actor instance (2.7 million actors per GB of heap) with a possible throughput of 50 million messages per sec on a single machine¹³. A sample push/pull communication with a throughput of 5000 messages per second amounts for 5% of CPU time. Therefore, the performance impact of the evaluation itself largely depends on the complexity of the assertion itself. The assertion would however need to be evaluated anyway, regardless if contracts are used or not—*i.e.* without contracts it would be an if-condition in the code.

Like in other languages, the runtime verification of contracts can be turned off leaving the operational methods unaffected (with all the consequences of unprotected code).

Reliability. Contracts impact both correctness and robustness properties [29]. Correctness is linked to contract efficiency—*i.e.* the ability of a contract to detect a failure. A contract violation implies the execution of an erroneous system. By injecting errors into the system and recording contract violations, it is possible to estimate their impact on software vigilance and diagnosability [26]. This approach is part of our plans for further work.

In addition to record state semantics, contracts also aid with bug assessment [30]. A violation of a precondition indicates a problem in the client while postcondition violation manifests a service failure.

In a recent study, Cámará *et al.* [8] made an experimental evaluation of the robustness of the Rainbow framework for architecture-based self-adaptation [18]. They defined a set of mutation rules, which were systematically applied to controller input in order to explore the limit conditions that according to the study are the typical sources of robustness issues. Essentially, the mutation rules feed

¹³ <http://bit.ly/1gHM975>.

the controller with invalid inputs, such as nulls, empty strings, wrong timestamps, or by overflowing or underflowing the input domain bounds. Their experiment uncovered robustness issues in about half of the tests they conducted, majority of which resulted in salient failures and a few even crash the controller. These results confirm the importance of having explicit assumption on inputs which in FCDL is supported by the behavioral contracts.

6.2 Limitations

While the use of contracts brings many advantages, they also come with some shortcomings. Contracts benefits are directly linked to their efficiency—*i.e.* what the contracts say and, often more importantly, what they do not, as they might give a false sense of correctness.

A low efficient contract only contributes to performance penalty instead of system reliability. In the case of poorly implemented contracts, such penalty might seriously affect the overall performance of the whole system. Furthermore, this can lead to an even worse situation when a contract is wrong. There are three types of incorrect contracts: (i) an inaccurate or false assertion that executes faulty system, (ii) a bug in the assertion throwing a runtime exception, and (iii) an impure function that violates the descriptive nature of contract assertions and mutates the component state upon evaluation. The last one in particular might cause a serious harm and is usually difficult to detect. Currently, in the assertion language, there is no support to detect an impure functions.

6.3 Discussion

The objective of FCDL is to provide engineers and researchers with a flexible abstraction that allows them to easily experiment with self-adaptation without the need to deal with low-level system implementation details. Including contracts into FCDL contributes to this goal by making this abstraction more rigid, yet without hindering its use. Concretely, they allow developers to precisely specify the assumptions about the component operational conditions, interactions and behavior in a systematic way with a simple, yet expressive programming language. One of the main benefits of the contracts is that it guides developers to make a clear distinction between what is an expected, what is a special, and what is an exceptional case and about how to handle them. Therefore, even though, there is an initial higher development effort, which could make some software engineers reluctant, according to our experience, as the system grows, the advantages of clear separation outweighs it. Moreover, the generative approach ensures that statically verifiable contracts—*i.e.* interaction contract, static interaction and structural invariants remain preserved at the implementation level. While the adoption of contracts fosters the reuse of adaptive elements in FCLs, we also contribute to improve the separation of concerns by isolating contract verification from failure handling. This approach supports the definition of custom repair strategies depending on the context of deployment of the software system.

7 Related Work

Our work is related to interaction specification, component contracts and self-adaptive software systems engineering.

7.1 Interaction Specification

To address the architecture underspecification, Cassou *et al.* [9] propose to enrich SCC architecture descriptions by annotating components with *interaction contracts* that precise their interactions. We extend this notion and make it applicable to FCDL. Concretely, our extension to the original proposal includes support for: (i) components with multiple output ports, (ii) multiports, (iii) composites including IC inference algorithm, (iv) optional interaction contracts, (v) interaction contract completion verification.

There are other formalisms that are commonly used to specify interactions between components or processes in distributed systems [3, 28] or in hierarchical component-based architectures [37]. However, since they offer full description of an interaction sequences, they require more expertise to use them properly. It is also much harder to enforce a complete automata behavior by the generated code while the interaction contracts remain preserved at the implementation level [36]. Finally, as interaction contracts capture the specific properties of data and demand driven communication, they were easier to tailor to FCDL than more general automata-based models.

7.2 Component Contracts

Beugnard *et al.* [6] describe four levels of contracts in component-based systems: syntactic, behavior (as defined by Meyer [29]), synchronous, and quality-of-service (QoS). The contracts we introduced into FCDL partially follows this hierarchy. The syntactic contracts include the adaptive element interface specification together with interaction contracts. The pre and postconditions with state invariants behave similarly as the contracts defined by Meyer with the small extension of contract dependency and custom error messages. Since in actor-oriented programming model there is no need to protect mutable state, we do not need to support synchronous contracts *per se*. The data flow synchronization is already covered by interaction contracts and we further include interaction invariants to enforce liveness properties through LTL formulae. Finally, the QoS contracts are partially supported through the runtime exceptions and their consequent supervision strategy. However, explicit QoS negotiation and component rewiring at runtime is not supported and restart reconfiguration is used instead. Negotiation strategies inspired by agent-based systems have been proposed for hierarchical component-based systems [10]. They could be envisaged in our context, but the relationship with the control models would need a thorough study.

The combination of several kinds of contracts have also been proposed for hierarchical components, with the coupling of executable assertions and temporal logic [13] and with some composition properties enabling the creation of a

composite contract [14]. Contrary to these approaches, our work is tailored to the feedback loop architecture, ensuring more properties on the data flow synchronization, framing the implementation while being more technology agnostic.

In LeTraon *et al.* [26] the authors propose a formal way to evaluate the impact of contracts on system vigilance and ability to detect bugs. Since the work considers Meyer's behavioral contract, it shall be usable for FCDL and thus we plan to incorporate it into the FCDL tool support.

7.3 Self-Adaptive Software Systems Engineering

There is a number of frameworks, middlewares and model-driven engineering approaches to self-adaptive systems engineering. They aim to reduce the design and implementation effort and provide a solid foundation for engineering of self-adaptive software systems (cf. surveys in Salehie and Tahvildari [41] or Villegas *et al.* [44]). However, they often target specific types of adaptation problems and require the use of certain adaptation mechanism (*e.g.* utility theory in Rainbow [18]) or are applicable to a single domain (*e.g.* mobile applications in MUSIC [40]) or technology (*e.g.* Java-based systems in StarMX [4]), thereby limiting their applicability with respect to the problem being addressed [39]. Furthermore, they do not particularly focus on control theoretical controllers.

FCDL on the other hand focuses primarily on the integration aspect of SASS engineering. It does not promote any particular control approach and instead provides an abstraction of feedback control loops in which various scenarios can be modeled with diverse control mechanisms. Having based the implementation on Akka and Xbase currently limits the adaptive element implementation to JVM languages. This might pose a problem for scenarios where the touchpoints need to interact with an API that is not accessible from Java nor JNI. However, thanks to the model-driven engineering approach, it is possible to target different actor runtimes and use Xbase to synthesize code to languages other than Java¹⁴. The increasing popularity of the actor model gives us a variety of different frameworks available in various programming languages¹⁵. This should allow for deploying the proposed solution on top of a wide range of systems.

While across this paper we have mostly followed a control theoretical approach, it should be equally possible to use Rainbow [18] instead of the integral controller. Rainbow could then take advantage from all the contracts introduced in this chapter, which should in turn improve its robustness in the experiments such as the one mentioned in the previous section.

There is also a large body of work that focuses on design and simulation of feedback control primarily for embedded systems, for example Ptolemy II [16]. Ptolemy II is an extensive framework for simulation of concurrent actor-oriented systems with the ability to combine heterogeneous models of computation. We follow a similar actor-oriented approach and our execution semantics is comparable to Ptolemy *Push-Pull* model of computation. Similarly to Rainbow, it should

¹⁴ <https://wiki.eclipse.org/Xbase>.

¹⁵ http://en.wikipedia.org/wiki/Actor_model.

be possible to create a Ptolemy-based controller that can be used in FCDL. The same applies to other tools that are often used by control theorists such as Matlab/Simulink¹⁶ or Modelica¹⁷.

8 Conclusion

In this chapter we have presented a design-by-contract extension into *Feedback Control Definition Language*, a domain-specific modeling language for integrating control mechanisms into software systems through external feedback control loops. The aim is to allow researchers and engineer to explicitly specify their assumptions about the operational condition of the different components that form feedback control loops.

The presented contracts support this by embedding elements of formal specification into the feedback control loop element definitions. Concretely, we have defined first-class language support to specify (i) behavioral contracts to assert component behavior through state invariants and pre and postconditions, (ii) interaction contracts to express allowed component interactions, and (iii) structural and temporal invariants to define architecture constraints as well as design and execution time interaction invariants. The temporal invariants are specified using linear temporal logic while the assertion of the other contracts use a Java-like expression language. Next to these contracts, we have also defined a first-class language support for systematic fault handling.

All the different type of contracts have been illustrated on a real-world adaptation scenario of web server QoS management control.

As for future work there are several opportunities for further extending the contracts specifications and validation of our approach: (i) explore the extend to which contracts could express causalities to ensure that a particular action always leads to a certain system state change; (ii) enable runtime modifications of invariants time thresholds; (iii) provide more insights into contracts evaluation by combining the mutation rules from Cámará *et al.* [8] and the quantifying approach proposed by Le Traon *et al.* [26], shall allow us to have an automate way to estimate the levels of vigilance and diagnosability of given set of contracts, and finally (iv) develop a tool support to better facilitate the use of the presented FCDL contracts.

Acknowledgments. This work is partially supported by the Datalyse project www.datalyse.fr and was previously supported by the ANR SALTY project under contract ANR-09-SEGI-012.

¹⁶ <http://www.mathworks.com/products/simulink/>.

¹⁷ <https://openmodelica.org/>.

References

1. Abdelzaher, T., Bhatti, N.: Web server QoS management by adaptive content delivery. In: 7th International Workshop on Quality of Service (1999)
2. Abdelzaher, T., Shin, K., Bhatti, N.: Performance guarantees for Web server end-systems: a control-theoretical approach. *IEEE Trans. Parallel Distrib. Syst.* **13**(1), 80–96 (2002)
3. de Alfaro, L., Henzinger, T.A.: Interface automata. In: ACM SIGSOFT Software Engineering Notes, vol. 26 (2001)
4. Asadollahi, R., Salehie, M., Tahvildari, L.: StarMX: a framework for developing self-managing Java-based systems. In: 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (2009)
5. Berekmeri, M., Serrano, D.: A Control Approach for Performance of Big Data Systems. In: Proceeding of the 2014 IFAC World Congress (2014)
6. Beugnard, A., Jézéquel, J.M., Plouzeau, N., Watkins, D.: Making components contract aware. *Computer* **32**(7), 38–45 (1999)
7. Brooks, F.P.: No silver bullet essence and accidents of software engineering. *Computer* **20**(4), 10–19 (1987)
8. Cámarra, J., de Lemos, R., Larangeiro, N., Ventura, R., Vieira, M.: Robustness evaluation of the rainbow framework for self-adaptation. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing (2014)
9. Cassou, D., Balland, E., Consel, C., Lawall, J.: Leveraging software architectures to guide and verify the development of sense/compute/control applications. In: 33rd International Conference on Software Engineering (2011)
10. Chang, H., Collet, P.: Fine-grained contract negotiation for hierarchical software components. In: 31th EUROMICRO-SEAA Conference - CBSE Track (2005)
11. Cheng, B.H.C., et al.: Software engineering for self-adaptive systems: a research roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) *Software Engineering for Self-Adaptive Systems*. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_1
12. Cheng, S.W., Garlan, D., Schmerl, B.: Evaluating the effectiveness of the Rainbow self-adaptive system. In: 4th ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (2009)
13. Collet, P., Ozanne, A., Rivierre, N.: Enforcing different contracts in hierarchical component-based systems. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, pp. 50–65. Springer, Heidelberg (2006). https://doi.org/10.1007/11821946_4
14. Collet, P., Malenfant, J., Ozanne, A., Rivierre, N.: Composite contract enforcement in hierarchical component systems. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 18–33. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77351-1_3
15. Efting, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., Hanus, M.: Xbase: implementing domain-specific languages for Java. In: Proceedings of the 11th International Conference on Generative Programming and Component Engineering (2012)
16. Eker, J., Janneck, J., Lee, E., Ludvig, J., Neuendorffer, S., Sachs, S.: Taming heterogeneity - the Ptolemy approach. *Proc. IEEE* **91**(1), 127–144 (2003)
17. Filieri, A., Hoffmann, H., Maggio, M.: Automated design of self-adaptive software with control-theoretical formal guarantees. In: Proceedings 36th International Conference on Software Engineering (2014)

18. Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P.: Rainbow: architecture-based self adaptation with reusable infrastructure. *IEEE Comput.* **37**(10), 46–54 (2004)
19. Hebig, R., Giese, H., Becker, B.: Making control loops explicit when architecting self-adaptive systems. In: Proceeding of the Second International Workshop on Self-Organizing Architectures (2010)
20. Hellerstein, J., Diao, Y., Parekh, S., Tilbury, D.: Feedback Control of Computing Systems. Wiley Online Library, Hoboken (2004)
21. Hellerstein, J.L.: Engineering autonomic systems. In: Proceedings of the 6th International Conference on Autonomic Computing (2009)
22. Hewitt, C.: Viewing control structures as patterns of passing messages. *Artif. Intell.* **8**(3), 323–364 (1977)
23. Holzmann, G.J.: Spin Model Checker, 1st edn. Addison-Wesley Professional, Boston (2003)
24. Kříkava, F.: Domain-Specific Modeling Language for Self-Adaptive Software System Architectures. Ph.D. thesis, University of Nice Sophia-Antipolis (2013)
25. Kříkava, F., Collet, P., France, R.B.: ACTRESS: domain-specific modeling of self-adaptive software architectures. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing (2014)
26. Le Traon, Y., Baudry, B., Jézéquel, J.M.: Design by contract to improve software vigilance. *IEEE Trans. Software Eng.* **32**(8), 571–586 (2006)
27. Lu, Y., Abdelzaher, T., Lu, C., Tao, G.: An adaptive control framework for QoS guarantees and its application to differentiated caching. In: 10th International Workshop on Quality of Service (2002)
28. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing, PODC 1987 (1987)
29. Meyer, B.: Applying ‘design by contract’. *Computer* **25**, 40–51 (1992)
30. Meyer, B.: Object-Oriented Software Construction (1997)
31. Meyer, B.: Toward more expressive contracts. *J. Object Oriented Program.* **13**(4) (2000)
32. Niz, D.D., Bhatia, G., Rajkumar, R.: Model-based development of embedded systems: the SysWeaver approach. In: 12th IEEE Real-Time and Embedded Technology and Applications Symposium (2006)
33. Object Management Group: OMG Object Constraint Language (OCL). Technical report, February 2014
34. Odersky, M.: Contracts for scala. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Rošu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 51–57. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_5
35. Oreizy, P., Rosenblum, D.S., Taylor, R.N.: On the role of connectors in modeling and implementing software architectures. Department of Information and Computer Science, University of California, Technical report (1998)
36. Parizek P., Plasil, F., Kofron, J.: Model checking of software components: combining Java PathFinder and behavior protocol model checker. In: 30th Annual IEEE/NASA Software Engineering Workshop (SEW-30) (2006)
37. Plasil, F., Visnovsky, S.: Behavior protocols for software components. *IEEE Trans. Software Eng.* **28**(11), 1056–1076 (2002)
38. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (1977)

39. Ramirez, A.J., Cheng, B.H.C.: Design patterns for developing dynamically adaptive systems. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (2010)
40. Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S., Lorenzo, J., Mamelli, A., Scholz, U.: MUSIC: middleware support for self-adaptation in ubiquitous and service-oriented environments. In: Proceedings of the 1st Workshop on Mobile, MobMid 2008 (2008)
41. Salehie, M., Tahvildari, L.: Self-adaptive software: landscape and research challenges. *ACM Trans. Auton. Adapt. Syst. (TAAS)* **4**(2) (2009)
42. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. *IEEE Softw.* **20**(5), 42–45 (2003)
43. Stolz, V., Bodden, E.: Temporal assertions using AspectJ. *Electron. Notes Theoret. Comput. Sci.* **144**, 109–124 (2006)
44. Villegas, N.M., Müller, H.A., Tamura, G., Duchien, L., Casallas, R.: A framework for evaluating quality-driven self-adaptive software systems. In: 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (2011)
45. Zhao, Y.: A model of computation with push and pull processing. Technical report, Technical Memorandum UCB/ERL M03/51, University of California, Berkeley (2003)

Synthesis of Distributed and Adaptable Coordinators to Enable Choreography Evolution

Marco Autili, Paola Inverardi, Alexander Perucci, and Massimo Tivoli^(✉)

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica,
Università dell'Aquila, L'Aquila, Italy

{marco.autili,paola.inverardi,massimo.tivoli}@univaq.it,
alexander.perucci@graduate.univaq.it

Abstract. Software systems are often built by composing together software services distributed over the Internet. Choreographies are a form of decentralized composition that models the external interaction of the participant services by specifying peer-to-peer message exchanges from a global perspective. Nowadays, very few approaches address the problem of actually realizing choreographies in an automatic way. Most current approaches are rather static and are poorly suited to the need of the Future Internet. In this chapter, we propose a method for the automatic synthesis of evolving choreographies. Coordination software entities are synthesized in order to proxy and control the participant services' interaction. When interposed among the services, coordination entities enforce the collaboration specified by the choreography. The ability to evolve the coordination logic in a modular way enables choreography evolution in response to possible changes. We illustrate our method at work on a running example in the domain of Intelligent Transportation Systems (ITS).

1 Introduction

The Future Internet [12] promotes a distributed computing environment that will be increasingly inhabited by a virtually infinite number of software services. Software systems will be more and more built by composing together software services distributed over the Internet. This calls for new networking paradigms, new service infrastructures and architectures, as well as flexible and dynamic composition mechanisms.

Today's service composition mechanisms are based mostly on service orchestration, a centralized approach to the composition of multiple services into a larger application. Orchestration works well in rather static environments with predefined services and minimal environment changes. These assumptions are inadequate in the Future Internet vision, in which many diverse service providers and consumers keep changing and cannot be coordinated through a centralized approach. In contrast, service choreography is a form of decentralized composition that models the external interaction of the participant services by specifying peer-to-peer message exchanges from a global perspective [3, 4].

The need for service choreography was recognized in BPMN2 (Business Process Model and Notation Version 2.0¹), the de facto standard for specifying choreographies, which introduced choreography-modeling constructs. BPMN2 *Choreography Diagrams* model peer-to-peer communication by defining a multi-party protocol that, when put in place by the cooperating parties, allows reaching the overall choreography objectives in a fully distributed way. In this sense, service choreographies differ significantly from service orchestrations, in which one stakeholder centrally determines how to reach an objective through cooperation with other services. Future software systems will not be realized by orchestration only; they will also require choreographies. Indeed, services will be increasingly active entities that, communicating peer-to-peer, proactively make decisions and autonomously perform tasks according to their own imminent needs and the emergent global collaboration.

When third-party participants are involved, usually black-box services to be reused, a key enabler for the actual realization of choreographies is the ability to automatically compose services, and dynamically perform *external coordination* of their interaction. However, in a distributed setting, obtaining the coordination logic required to realize a choreography is nontrivial and error prone. As matter of fact, automatic support for realizing choreographies is needed.

As will be discussed in more detail in Sect. 6, nowadays, very few approaches address the problem of actually realizing choreographies in an automatic way. Current approaches are rather static and are poorly suited to the need of the Future Internet, which is a highly dynamic networking environment that brings together heterogeneous services ranging from business- to thing-based services. Moreover, choreography-based software systems may be in operation for a long time, and it is impractical (if not infeasible) to replace or retry the whole choreography process whenever a change occurs. Instead, choreographies should continuously evolve to meet modifications in the choreography specification, as well as changing execution context, e.g., to support new technologies.

Towards addressing these challenges, in this chapter we propose an approach to the automatic synthesis of evolving choreographies. The synthesis processor takes as input a BPMN2 choreography diagram together with a set of services discovered as possible candidates to play the choreography roles, and automatically generates a set of coordination software entities. When interposed among the services according to a predefined architectural style, these software entities act as proxies of the participant services to coordinate their interaction, when needed. Specifically, coordination entities (called *Coordination Delegates* - CDs) enforce the collaboration specified by the choreography diagram through distributed protocol coordination [4].

Following key principles of autonomic computing [18, 22] and related architecture concepts [19], it is widely recognized that, in order to effectively design adaptive systems, feedback loops enabling adaptiveness must become first-class entities [1, 5, 8, 9, 11, 24]. Furthermore, the system engineering process must be rethought in order to break the traditional division among development phases

¹ www.omg.org/spec/BPMN/2.0.

by moving some activities from design-time to deployment- and run-time, hence asking for the exploitation of models at run time [1, 9, 11].

Inspired by this valuable work in the literature, our CDs are first-class coordination entities. CDs represent external controllers that realize *multiple interacting feedback loops* enabling choreography evolution at the level of both the supervised participants (local evolution) and the emergent collaboration among them (global evolution). In this sense, the choreography-based systems we target are autonomic systems where individual autonomic elements, i.e., CDs manage their internal status and behavior and their relationships with the other autonomic elements in accordance with the choreography specification.

The method presented in this chapter thoroughly advances our previous work [3, 4] by enhancing the synthesis method to also deal with the dynamic evolution of the coordination logic implied by the choreography in response to choreography specification and context changes.

The chapter is structured as follow. Section 2 introduces a running example in the ITS domain. Section 3 overviews our choreography synthesis method. Section 4 clarifies what is the meaning of choreography evolution in our setting. Furthermore, it describes the way we realize multiple interacting feedback loops to enable choreography evolution in response to goal changes. Section 5 describes our method at work on the running example. Section 6 discusses related work, and conclusions and future work are given in Sect. 7.

2 Running Example

The example introduced in this section concerns a portion of an eco-friendly route guidance system for assisting drivers during their trip. The considered portion of the system is designed as a choreography of a number of software services. The choreography specification shown in Fig. 1 is given by using BPMN2 Choreography Diagrams. In BPMN2, a choreography *task* (e.g., **Send Origin and Destination**) is an atomic activity that represents an interaction by means of one or two (request and optionally response) message exchanges between two participants (e.g., ND and BS-MAP). Graphically, BPMN2 choreography diagrams uses rounded-corner boxes to denote choreography tasks. Each of them is labeled with the roles of the two participants involved in the task, and the name of the service operation performed by the initiating participant and provided by the other one. A role contained in the white box denotes the initiating participant. In particular, the BPMN2 standard specification employs the theoretical concept of a token that, traversing the sequence flows and passing through the tasks specified by the choreography, aids to define its behaviors. The start event generates the token that must eventually be consumed at an end event. Basically, among other details, the BPMN2 choreography diagram in Fig. 1 specifies that the task **Route Request** is followed by the task **Routes Suggestion**.

The participants involved in the choreography specified by the diagram in Fig. 1 are:

- ND represents the driver's client app provided by the navigation device of the driver. DTS-GOOGLE represents a service that provides road traffic information: Google Maps in our example.
- DTS-HERE represents a service that provides road traffic information: NOKIA HERE Maps in our example.
- BS-MAP represents a service that supports ND while accessing digital maps. Map information can be both local and on-demand through Internet access. BS-MAP is used for accessing the Google Maps and NOKIA HERE Maps services.
- TRVC represents one of the services offered by Trafikverket² (the Swedish national road administration) that allows for collecting route-related information (e.g., traffic load, road conditions, accidents, etc.) from different sources.
- DTS-TRV-ACC represents a service that provides up-to-date accident related information.

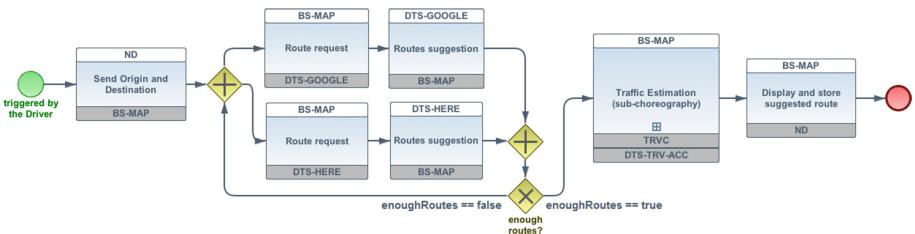


Fig. 1. Choreography specification of the running example

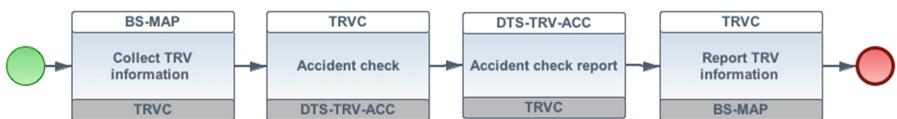


Fig. 2. Traffic estimation sub-choreography

The choreography is triggered by the driver upon inserting both origin and destination through the Navigation Device (ND). After receiving origin and destination information from ND, the BS-MAP service invokes in parallel the services DTS-GOOGLE and DTS-HERE (see the parallel branch represented as an rhombus marked with a “+” with two outgoing arrows). These services both serve to

² <http://www.trafikverket.se/en/>.

retrieve available routes from the origin to the destination. When done, the related parallel flows are joined to synchronize on the production of the set of available routes (see the merging branch represented as an rhombus marked with a “+” with two incoming arrows). Upon receiving the set of available routes, BS-MAP determines if they are enough (see the conditional branching represented as a rhombus marked with a “x”): (i) in the positive case, it triggers the (simplified version of the) Traffic Estimation sub-choreography to estimate the traffic situation by checking the presence of accidents along the determined routes (see Fig. 2); (ii) in the negative case, the BS-MAP service re-invokes DTS-GOOGLE and DTS-HERE to determine other routes. When traffic information is calculated for each route, these are displayed into the ND in order to suggest the most eco-friendly routes to the driver. Then, the choreography we are considering here terminates.

For the sake of presentation, in Figs. 1 and 2, input/output messages of tasks are not shown although they are internally stored according to their XML Schema type definition in the WSDL.

To clarify the kind of coordination issues that can arise when realizing choreography-based systems by reusing existing services, let us consider the following scenario. An existing BS-MAP service to be reused and bound to the BS-MAP participant could be reasonably implemented in a way that is able to suitably interact with both the concrete services playing the roles of DTS-GOOGLE and DTS-HERE, but without realizing the join specified by the choreography. This means that when left “uncoordinated”, BS-MAP may never receive enough routes, hence never displaying suggested routes to ND. From a technical point of view, this clearly represents an undesired interaction. In principle, as usual in the practice of concurrent systems development, one could think of solving this issue by exploiting timeouts. However, this would mean that BS-MAP may attempt at displaying and store only a subset of all possible suggested routes to ND, e.g., the ones suggested by DTS-GOOGLE, hence estimating the traffic only for this subset of routes. From the user point of view, this is an undesired interaction since it might result in an unsatisfactory route suggestion that does not account for possible further routes with better traffic conditions. As a consequence, this may lead the user to not trust the choreography-based application anymore.

Furthermore, as far as choreography evolution is concerned, the specified choreography can vary depending on context changes due to, e.g., services availability and user preferences. These changes impact the choreography specification that has to evolve accordingly, hence affecting also the corresponding coordination logic. For instance, the Traffic Estimation sub-choreography could be realized differently from what is shown in Fig. 2 depending on context conditions that are dynamically evaluated and trigger choreography evolution. This could be the case when, e.g., alternative traffic information collector services are available with respect to the considered TRVC and DTS-TRV-ACC.

Our synthesis method, as overviewed in Sect. 3, is able to automatically deal with these kinds of distributed coordination and evolution issues. Sections 4 and 5 describe the method at work on the running example.

3 Method Description

As already introduced, the synthesis processor takes as input a BPMN2 choreography diagram together with a set of services discovered as possible candidates to play the choreography roles, and automatically generates a set of coordination entities (i.e., CDs). Figure 3 pictorially describes the main steps of the synthesis process.

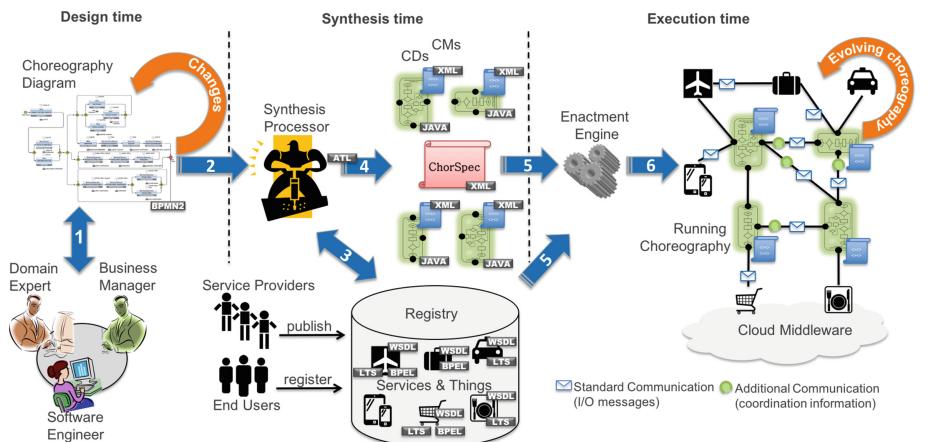


Fig. 3. From choreography design to execution, through automatic synthesis

Step 1. Software producers cooperate with, e.g., domain experts and business managers to

- set the business goal (for example, assist travellers from arrival, to staying, to departure),
- identify the tasks and participants required to achieve the goal (for example, reserving a taxi from the local taxi company, purchasing digital tickets at the train station, and performing transactions through services based on near-field communication in a shop), and
- specify how participants must collaborate through a BPMN2 choreography diagram.

Step 2. The synthesis processor takes as input the BPMN2 choreography diagram.

Step 3. The synthesis processor queries the registry to discover services suitable for playing the choreography's roles. The registry contains services published by

providers (for example, transportation companies and airport retailers) that have identified business opportunities in the domain of interest. As service interfaces description, the synthesis processor assumes WSDL³. To describe service interaction behavior (i.e., the specification of the flow of messages exchanged with the environment), the synthesis processor assumes an automata-based specification, i.e., a Labeled Transition System – LTS, or a BPEL specification⁴. The registry also contains the registration of users interested in exploiting the choreography through their mobile apps.

Step 4. Starting from the choreography diagram and the set of discovered (or specific⁵) services, the synthesis processor generates the set of CDs. The processor also generate the so called **ChorSpec**, a specification to be used by the Enactment Engine (EE) component for deploying and enacting the choreography. The **ChorSpec** is an XML-based declarative description of the choreography that specifies the locations of the services, and their interdependencies with the generated CDs. The synthesis exploits model transformations. The transformations are implemented through ATL⁶, a domain-specific language for realizing model-to-model (M2M) transformations. ATL transformations comprise a number of rules, each of which manages a specific BPMN2 modeling construct.

Step 5. The generated CDs, together with the description of the services and their dependencies, serve as input to the EE for deployment and enactment. The description of the EE is outside the scope of this chapter. Interested readers can refer to [26] for details.

Step 6. Following the dependencies, CDs are then interposed among the participant services needing coordination.

More technically, when interposed among the services according to a predefined architectural style (a sample instance of which is shown in Fig. 4), CDs act as proxies of the participant services to coordinate their interaction, when needed. CDs guarantee the collaboration specified by the choreography specification through distributed protocol coordination [4]. CDs perform pure coordination of the services' interaction (i.e., *standard communication* in the figure) in a way that the resulting collaboration realizes the specified choreography. To this purpose, the coordination logic is extracted from the BPMN2 choreography diagram and is distributed among a set of *Coordination Models* (CMs) that codify coordination information. Then, at runtime, the CDs manage their CMs and exchange this coordination information (i.e., *additional communication*) to prevent possible *undesired interactions*. As explained below, the latter are those interactions that do not belong to the set of interactions allowed by the choreography specification and can happen when the services collaborate in

³ Web Services Description Language, www.w3.org/TR/wsdl.

⁴ Web Services Business Process Execution Language, <https://www.oasis-open.org/committees/wsbpel>.

⁵ If possible candidate services are specified directly by the user of the synthesis processor, the service discovery step can be skipped.

⁶ Atlas Transformation Language www.eclipse.org/atl.

an uncontrolled way. The coordination logic embedded in CDs is obtained by a distributed coordination algorithm implemented in Java; each CD runs its own instance of the algorithm. Once deployed by the EE, CDs support the correct execution of the choreography by realizing the required distributed coordination logic among the participant services. Details on the distributed coordination algorithm can be found in [4].

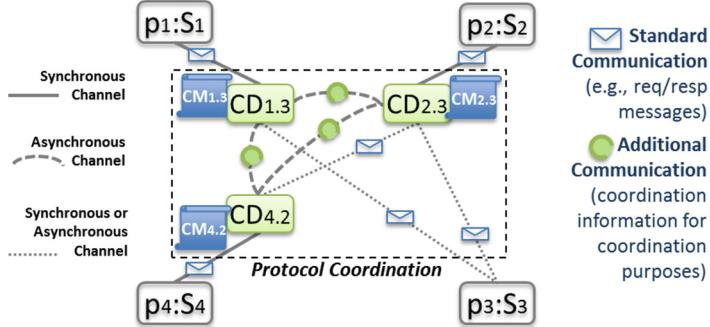


Fig. 4. Architectural style (a sample instance of)

In summary, at deployment time, for each interface that a service S_i requires from another service S_j , a $CD_{i.j}$ is interposed between the S_i and S_j that play the roles of the choreography participants p_i and p_j , respectively. Furthermore, the Service-CD channel to exchange standard communication is synchronous; the CD-CD channel to exchange standard communication is either synchronous or asynchronous depending on the services' implementation; the CD-CD channel to exchange additional communication (i.e., coordination information) is asynchronous.

4 Dealing with Choreography Evolution

As already introduced, the ability to evolve the coordination logic in order to enable choreography evolution in response to possible changes is a key factor of our method. Our methods concentrates on two types of changes, namely *choreography specification changes* and *context changes* discussed below.

We exploit the *BPMN2 Call Choreography* construct to express, at design time, choreography variation points and associated context conditions. At run time, each variation is enabled/disabled depending on whether the associated context conditions are satisfied or not.

Choreography specification changes – The source of this type of changes are the choreography modelers (i.e., software producers, domain experts, and business managers) that modify the current specification of the choreography

at run time (e.g., to meet new business/customers needs). The changes that choreography modelers can perform are restricted to the extent of the already specified Call Choreographies. This means that the modeler can add/remove variation points, or modify existing ones. Note that restricting the way choreography modelers can change the choreography specification is a reasonable choice since completely altering the specification may “distort” the business-level goal the choreography was originally specified for. As a consequence, a complete alteration of the specification naturally requires to re-synthesize the choreography from scratch. On the practical side, this allows for effectively coping with choreography evolution in an automatic way. Furthermore, an interesting observation is that the granularity of changes is per single tasks, when meaningful according to the specification of Call Choreographies.

Context changes – The source of this type of changes is the dynamically “sensed” choreography context. At run time, context changes drive the activation/deactivation of variation points, provided that the specified context conditions are exclusive. That is, only one variation can be enabled at run time and the execution of a variation is atomic, and no other variations can be enabled while another variation is being executed. A *ChoreographyContext* is characterized by conditional expressions over “facts” that has to be evaluated at run time. On the practical side, each Call Choreography requires a suitable *ContextEvaluator* to allow the run-time check of the context conditions specified for each variation in the Call Choreography. Within the method presented in this chapter, a choreography context distinguishes among *MessageContext*, *ExecutionContext*, and *DomainApplicationContext*.

- A message context is described in terms of data contained in the messages exchanged among the involved services up to a given point in time.
- An execution context is described by using information about possible computational resources. Thus, in order to “sense” an execution context, dedicated functionalities should be provided to get information about the resources offered by the execution environment, e.g., CPU and memory utilization, network connectivity.
- A domain/application context is characterized by conditions on the surrounding environment, given that dedicated functionalities (e.g., supported by dedicated sensors) are provided to sense it.

Overall, the choreography context enables choreography evolution through appropriate adaptation actions, for instance, switching among choreography variations as well as the replacement of a disconnected or low-performing service by another, or reserving on-demand additional computing resources for choreography peers.

By continuing our running example, Figs. 5, 6, 7 and 8 shows a slightly modified version of the choreography specification shown in Figs. 1 and 2. Three variation points are specified (see Figs. 6, 7 and 8), hereafter referred to as V1, V2, and V3. Furthermore, the ND app allows the driver to also specify a user preference concerning the accuracy of the traffic estimation that, together with service availability, drives the selection of the specified choreography variations.

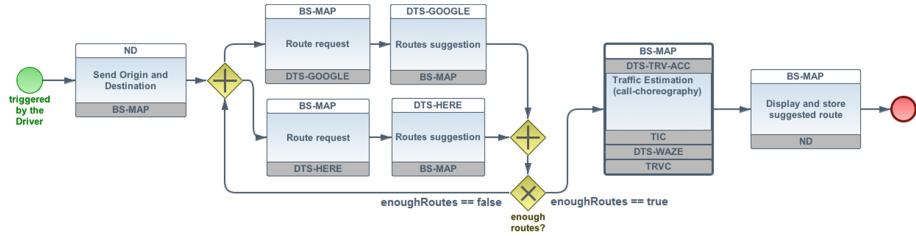


Fig. 5. Choreography specification with Variations

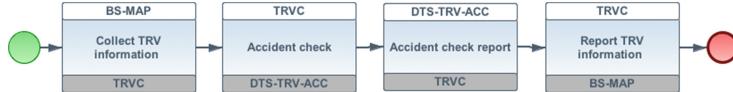


Fig. 6. Traffic Estimation Variation V1

As shown in Fig. 6, variation V1 is exactly the same as the sub-choreography previously shown in Fig. 2 and already discussed in Sect. 2. The choreography context specified for V1 is an execution context plus a message context. The execution context conditions predicate on the availability of two software computing resources, i.e., two services – say TRVC and DTS-TRV-ACC – that can play the roles of TRVC and DTS-TRV-ACC, respectively. The message context conditions predicate on the values of the `estimationAccuracy` parameter contained in the `sendOriginAndDestinationRequest` message internally associated to the task `Send Origin and Destination`. It is an enumerated type whose values range over `low`, `medium`, and `high`. When TRVC and DTS-TRV-ACC are available, and `estimationAccuracy == low`, then V1 will be enabled. This means that the Call Choreography in Fig. 5 is realized through the execution of V1.

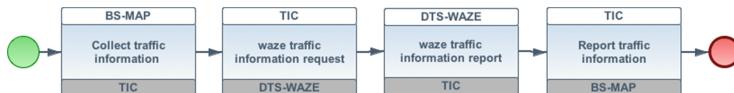


Fig. 7. Traffic Estimation Variation V2

As shown in Fig. 7, variation V2 describes a choreography involving BS-MAP and two new services, namely TIC and DTS-WAZE. The former is an alternative traffic information collector service with respect to TRVC. The latter is a service for retrieving road traffic information provided by the crowd sourcing application Waze⁷. V2 is an alternative choreography with respect to V1 that can be executed to perform traffic information estimation. The difference is that V1 estimates traffic information by checking the presence of accidents, whereas

⁷ www.waze.com.

V2 estimates traffic by accounting for the real-time traffic and road information shared among other drivers in the same area. Analogously to V1, the V2 context predicates on the availability of TIC and DTS-WAZE, plus checking that `estimationAccuracy == medium`.

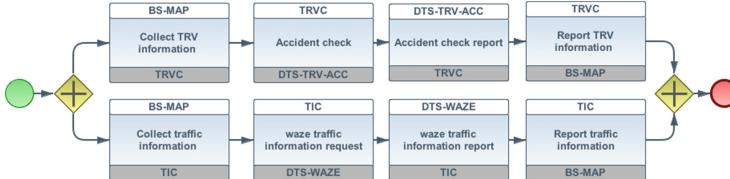


Fig. 8. Traffic Estimation Variation V3

As shown in Fig. 8, variation V3 executes two parallel flows corresponding to the sequential flows in V1 and V2. This choreography estimates traffic by combining in parallel the traffic estimation logic of both V1 and V2. Thus, V3 is able to consider information concerning the presence of accidents in conjunction with the real-time information coming from different drivers. On the practical side, V3 is a more accurate traffic estimation choreography. The V3 context predicates on the availability of TRVC, DTS-TRV-ACC, TIC, and DTS-WAZE, plus checking that `estimationAccuracy == high`.

5 Method at Work

This section describes how our synthesis method automatically synthesizes the required CDs, and related CMs, for the previously introduced running example. As described in the previous section, these CDs account for possible dynamic evolution of the choreography characterized by modeling choreography variation points. However, in order to ease the understandability of each step of the synthesis method, for now, let us consider the choreography shown in Fig. 1 that does not specify variation points.

Our running example involves services that are all both providers and consumers (i.e., they are prosumers) and, as such, requires the generations of ten CDs, each coupled with its own CM (Fig. 9). Note also that CDs are not required for any possible pair of participants, rather they are generated only for pairs of participants involved in the same task. This means that we do not require to always keep the global state of the choreography to enforce it. This characterizes the distributed nature of our synthesis approach. As already introduced, when interposed between the services to be choreographed, CDs coordinate the interaction of the involved services in a fully distributed way. The coordination logic is distributed among the related CMs that codify coordination information. CMs are not derived directly from the BPMN2 choreography specification, rather they are derived passing through the intermediate CeFM model.

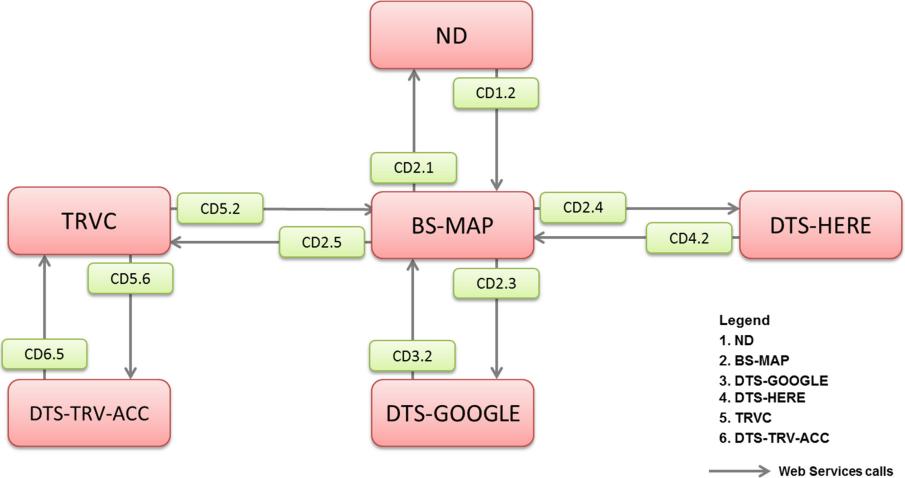


Fig. 9. Architecture of the running example with CDs

CeFM generation – A CeFM is a choreography model that, conforming to the BPMN2 standard specification, makes explicit coordination-related information that in BPMN2 is implicit. This allows to statically infer the information needed for enabling distributed coordination that, otherwise, should be calculated at run time for each choreography instance and for each execution of it. For instance, the CeFM model specifies the source and target state from which a task is initiated and terminated, the corresponding transition and enabling condition. The synthesis processor implements a model-based transformation to automatically generates the CeFM model out of the BPMN2 choreography specification. Figure 10 shows the CeFM automatically derived out of the choreography diagram in Fig. 1 and the sub-choreography diagram in Fig. 2. The states in the CeFM are automatically assigned a number by the performed model-based transformation. Transitions are labeled by the initiating participant, the performed task, and the receiving participant, separated by ‘.’. Transitions with no labels (e.g., see the light-gray transition from S2 to S3) are the so called ε -transitions representing “internal” steps. The latter are introduced for dealing with non-plain states (e.g., S3, S10 and S12 in the figure) both structurally and semantically. For instance, they can be used to specify cascading fork states or to model the (internal) event of creating parallel flows, which originate from a fork state.

The initial state of a CeFM (e.g., the filled state in Fig. 10) generates the token that must eventually be consumed at the final state (e.g., the double-circled state). If a token is on a plain state (e.g., S1), then the outgoing flow is ready to be executed hence performing the operation characterized by its task label. If a token is on an alternative state (e.g., S12) then one of the outgoing flows, whose (internally associated) conditional expression evaluates to true, is ready to be

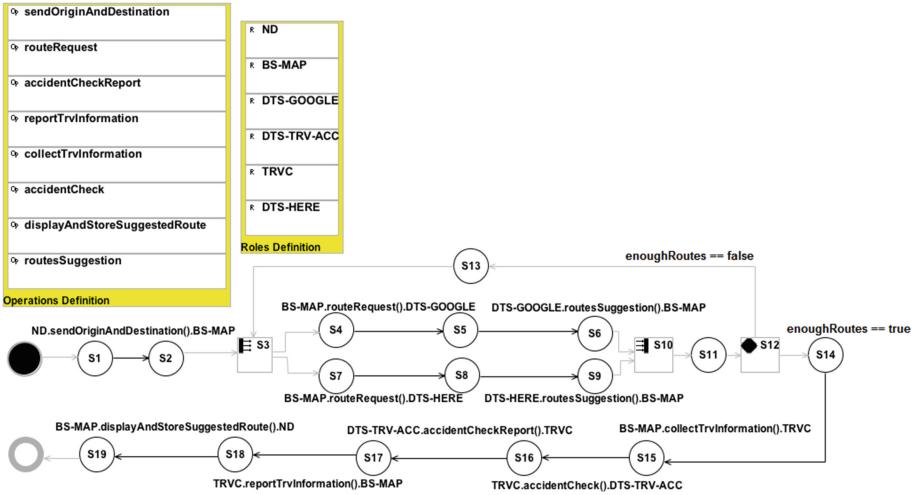


Fig. 10. CeFM model of the running example

executed. If a token is on a loop state, then the outgoing flow leading to the loop entry state is ready to be executed if its guarding conditional expression is true. Otherwise, the flow leading to the exit state will be performed. If a token is on a fork state (e.g., S3), then all outgoing flows are taken simultaneously leading to several parallel executions, each having its own token. Concerning join states (e.g., S10), if for each incoming flow there is a token, then the outgoing flow is triggered.

It is worth noticing that the CeFM is independent from the specific modeling notation used to specify choreographies. This means that the our synthesis method can be used in different practical contexts in which different modeling notations might be adopted for specifying choreographies, provided that a dedicated model-to-model transformation into CeFM is developed.

CDs and CMs generation – After the CeFM has been automatically generated, our synthesis method decomposes it into a set of CMs, one for each CD. Informally, each CM represents a local view of the specified choreography. It is local since it models the coordination logic that has to be enforced on the interaction between two participants, p_i and p_j . This is done by exchanging coordination information among the coordination delegate $CD_{i,j}$ – supervising p_i by proxying p_j – and the CDs it needs to synchronize with. In this acceptation, the set of all CMs can be considered as a distributed model of the specified choreography.

A CM codifies the coordination information that the associated CD has to interpret in order to cooperatively realize the specified choreography by interacting with the other CDs. Figure 11 shows a representation of all the CMs generated from the CeFM in Fig. 10. For instance, $CM_{ND.BS-MAP}$ is the coordination information model of $CD_{ND.BS-MAP}$ that coordinates (from outside)

the interaction between the two services that will play the roles ND and BS-MAP, respectively. Thus, $CD_{ND.BS-MAP}$ is a proxy of BS-MAP that provides a set of WSDL operations whose names can be automatically derived by the labels of the tasks involving ND and BS-MAP. More precisely, the name of a CD operation is automatically produced by concatenating the words in the corresponding task label and using the *camelCase* naming convention (i.e., the name of an operation should usually start with a lowercase sequence of characters). For instance, specifically considering the task `Send Origin and Destination`, $CD_{ND.BS-MAP}$ provides the operation `sendOriginAndDestination(origin, destination)`.

Technically, a CM is an XML file codifying a set of tuples. A CM's tuple is of the form $\langle s, t, s', CD_{s'}, \rho, Notify_s, Wait_s \rangle$ where:

- s denotes the CeFM source state from which the related CD can either perform the operation t or take a move without performing any operation (i.e., the CD can step over an ε -task). In both cases, s' denotes the reached target state.
- $CD_{s'}$ contains the set of (identifiers of) those CDs whose supervised services became active in s' , i.e., the ones that will be allowed to require/provide some operation from s' . This information is used by the “currently active” CDs to inform the set of “to be activated” CDs (in the target state s') about the changing global state.
- ρ is a conditional expression whose run-time evaluation is performed to select the correct flow(s) in the CeFM.
- $Notify_s$ contains the predecessors of a join state that a CD, when reaching it, must notify to the other CDs in the parallel flow(s) of the same originating fork. Complementary, $Wait_s$ contains the predecessors of join states that must be waited for.

Intuitively, the first tuple in $CM_{ND.BS-MAP}$ specifies that $CD_{ND.BS-MAP}$ can perform the operation `sendOriginAndDestination` from the source state $S1$ to the target state $S2$; whereas, the second tuple specifies that $CD_{ND.BS-MAP}$ can step over $S2$ and reach $S3$ from where two parallel flows can be undertaken. That is, as specified by the third and fourth tuple, $CD_{ND.BS-MAP}$ activates $CD_{BS-MAP.DTS-GOOGLE}$ on $S4$ and $CD_{BS-MAP.DTS-HERE}$ on $S7$.

Considering the second tuple $\langle S6, \epsilon, S10, \{\}, true, \{[S6, (DTS-HERE.BS-MAP)], [S9, (DTS-HERE.BS-MAP)]\} \rangle$ in $CM_{DTS-GOOGLE.BS-MAP}$, $CD_{DTS-GOOGLE.BS-MAP}$ notifies $S6$ to $CD_{DTS-HERE.BS-MAP}$ and waits for receiving the notification from $CD_{DTS-HERE.BS-MAP}$ about $S9$. Complementarily, considering the second tuple $\langle S9, \epsilon, S10, \{\}, true, \{[S9, (DTS-GOOGLE.BS-MAP)], [S6, (DTS-GOOGLE.BS-MAP)]\} \rangle$ in $CM_{DTS-HERE.BS-MAP}$, $CD_{DTS-HERE.BS-MAP}$ notifies $S9$ to $CD_{DTS-GOOGLE.BS-MAP}$ and waits for receiving the notification from $CD_{DTS-GOOGLE.BS-MAP}$ about $S6$. Note that the same considerations apply in case different parallel threads of the same CD are involved in a join state.

Then, considering the fifth and sixth tuples of $CD_{DTS-GOOGLE.BS-MAP}$, this CD can reach either $S13$ or $S14$ according to the evaluation of the related

CM_{IND}.BS-MAP	CM_{BS-MAP.ND}
$\langle S1, sendOriginAndDestination(), S2, \{\}, true, \{\}, \{\} \rangle$	$\langle S18, displayAndStoreSuggestedRoute(), S19, \{\}, true, \{\}, \{\} \rangle$
$\langle S2, e, S3, \{\}, true, \{\}, \{\} \rangle$	$\langle S19, \varepsilon, FINAL, \{\}, true, \{\}, \{\} \rangle$
$\langle S3, e, S4, \{(BS-MAP.DTS-GOOGLE)\}, true, \{\}, \{\} \rangle$	
$\langle S3, e, S7, \{(BS-MAP.DTS-HERE)\}, true, \{\}, \{\} \rangle$	
CM_{BS-MAP.DTS.GOOGLE}	CM_{BS-MAP.DTS.HERE}
$\langle S4, routeRequest(), S5, \{(DTS-GOOGLE.BS-MAP)\}, true, \{\}, \{\} \rangle$	$\langle S7, routeRequest(), S8, \{(DTS-HERE.BS-MAP)\}, true, \{\}, \{\} \rangle$
CM_{DTS}.BS-MAP	CM_{DTS}.HERE.BS-MAP
$\langle S5, routeSuggestion(), S6, \{\}, true, \{\}, \{\} \rangle$	$\langle S8, routesSuggestion(), S9, \{\}, true, \{\}, \{\} \rangle$
$\langle S6, e, S10, \{\}, true, \{\}, \{\} \rangle$	$\langle S9, \varepsilon, S10, \{\}, true, \{\}, \{\} \rangle$
$\{ [S6, \{DTS-HERE, BS-MAP\}],$	$\{ [S9, \{DTS-GOOGLE, BS-MAP\}],$
$[S9, \{DTS-HERE, BS-MAP\}] \}$	$[S6, \{DTS-GOOGLE, BS-MAP\}] \}$
$\langle S10, e, S11, \{\}, true, \{\}, \{\} \rangle$	$\langle S10, \varepsilon, S11, \{\}, true, \{\}, \{\} \rangle$
$\langle S11, e, S12, \{\}, true, \{\}, \{\} \rangle$	$\langle S11, \varepsilon, S12, \{\}, true, \{\}, \{\} \rangle$
$\langle S12, e, S13, \{\}, enoughRoutes == false, \{\}, \{\} \rangle$	$\langle S12, \varepsilon, S13, \{\}, enoughRoutes == false, \{\}, \{\} \rangle$
$\langle S12, e, S14, \{(BS-MAP.TRVC)\}, enoughRoutes == true, \{\}, \{\} \rangle$	$\langle S12, \varepsilon, S14, \{(BS-MAP.TRVC)\}, enoughRoutes == true, \{\}, \{\} \rangle$
$\langle S13, e, S3, \{\}, true, \{\}, \{\} \rangle$	$\langle S13, \varepsilon, S3, \{\}, true, \{\}, \{\} \rangle$
$\langle S3, e, S4, \{(BS-MAP.DTS-GOOGLE)\},$	$\langle S3, \varepsilon, S4, \{(BS-MAP.DTS-GOOGLE)\}, true, \{\}, \{\} \rangle$
$\langle S3, e, S7, \{(BS-MAP.DTS-HERE)\}, true, \{\}, \{\} \rangle$	$\langle S3, \varepsilon, S7, \{(BS-MAP.DTS-HERE)\}, true, \{\}, \{\} \rangle$
CM_{BS-MAP.TRVC}	CM_{TRVC.DTS-TRV-ACC}
$\langle S14, collectTrvInformation(),$	$\langle S15, accidentCheck(),$
$S15, \{(TRVC.DTS-TRV-ACC)\}, true, \{\}, \{\} \rangle$	$S16, \{(DTS-TRV-ACC.TRVC\}, true, \{\}, \{\} \rangle$
CM_{DTS-TRV-ACC.TRVC}	CM_{TRVC.BS-MAP}
$\langle S16, accidentCheckReport(), S17, \{(TRVC.BS-MAP\}, true, \{\}, \{\} \rangle$	$\langle S17, reportTrvInfo(), S18, \{(BS-MAP.ND\}, true, \{\}, \{\} \rangle$

Fig. 11. CMS tuples derived from the CeFM in Fig. 10

conditions, i.e., $\text{enoughRoutes} == \text{false}$ or $\text{enoughRoutes} == \text{true}$, respectively. This means that, after the operation routesSuggestion has been requested by DTS-GOOGLE and forwarded to BS-MAP, in the case the conditional expression $\text{enoughRoutes} == \text{true}$ holds, $CD_{DTS-GOOGLE.BS-MAP}$ uses the sixth tuple $\langle S12, \varepsilon, S14, \{\text{BS-MAP.TRVC}\}, \text{enoughRoutes} == \text{true}, \{\}, \{\} \rangle$ to step over the loop state $S12$, reach $S14$, and inform $CD_{BS-MAP.TRVC}$ about the new global state $S14$.

Summing up, at run time, CDs exchange the coordination information codified into the CMs to prevent undesired interactions, i.e., those interactions that do not belong to the set of interactions allowed by the choreography specification and can happen when the services collaborate in an uncontrolled way.

To illustrate the distributed coordination enforced by the CDs at work on our example, Fig. 12 shows a sequence diagram representing an excerpt⁸ of the exchange of business- and coordination-level messages among the participant services and their supervising CDs. As it is evident from the diagram, the collaboration of the synthesized CDs coordinates the interaction among the services in order to let them perform only the interactions specified by the CeFM, hence preventing the undesired interaction(s) mentioned above.

The coordination logic that the CD performs by relying on its CM is an implementation of a distributed coordination algorithm. The latter inherits the distributed mutual exclusion principle and leverages some foundational notions (such as happened-before relation, partial ordering, and time-stamps) of the algorithm in [25]. More precisely, the enforcement of mutual exclusive access to a single resource in [25] is scaled up to the enforcement of concurrent task flows according to arbitrarily complex choreographies. A CD is a *reactive* entity that, at each iteration of the algorithm, WAITS for receiving either a request from the service it supervises or NOTIFY/UPDATE messages from the other CDs. The most important aspect here is that, upon reaching a join state, the *involved* CDs NOTIFY and WAIT for each other in order to synchronize their execution. At run time, NOTIFY messages (together with a simple notion of *priority* initially associated to each CD) are used to realize join states as distributed synchronization points. Then, when a join state has been realized, and hence all the parallel executions have been synchronized, the CD with highest priority is in charge of notifying the CDs that, according to the choreography, are allowed to proceed. Each $CD_{i,j}$ sends NOTIFY messages according to the coordination information contained in the sixth element of the $CM_{i,j}$ tuples. The “co-related” WAIT primitive is instead performed according to the information contained in the seventh element of the tuples. In our implementation, the correct co-relation among WAIT primitives and NOTIFY (or UPDATE) messages is realized by means of dedicated queues that, for each WAIT, buffer the expected NOTIFYs (or UPDATES). $CD_{i,j}$ uses UPDATE messages whenever the current global state of the CeFM changes according to the performed coordination. By considering the coordination information represented by the fourth element of its tuples,

⁸ From the initial state to $S15$, and by assuming that the conditional expression $\text{enoughRoutes} == \text{true}$ is evaluated to true.

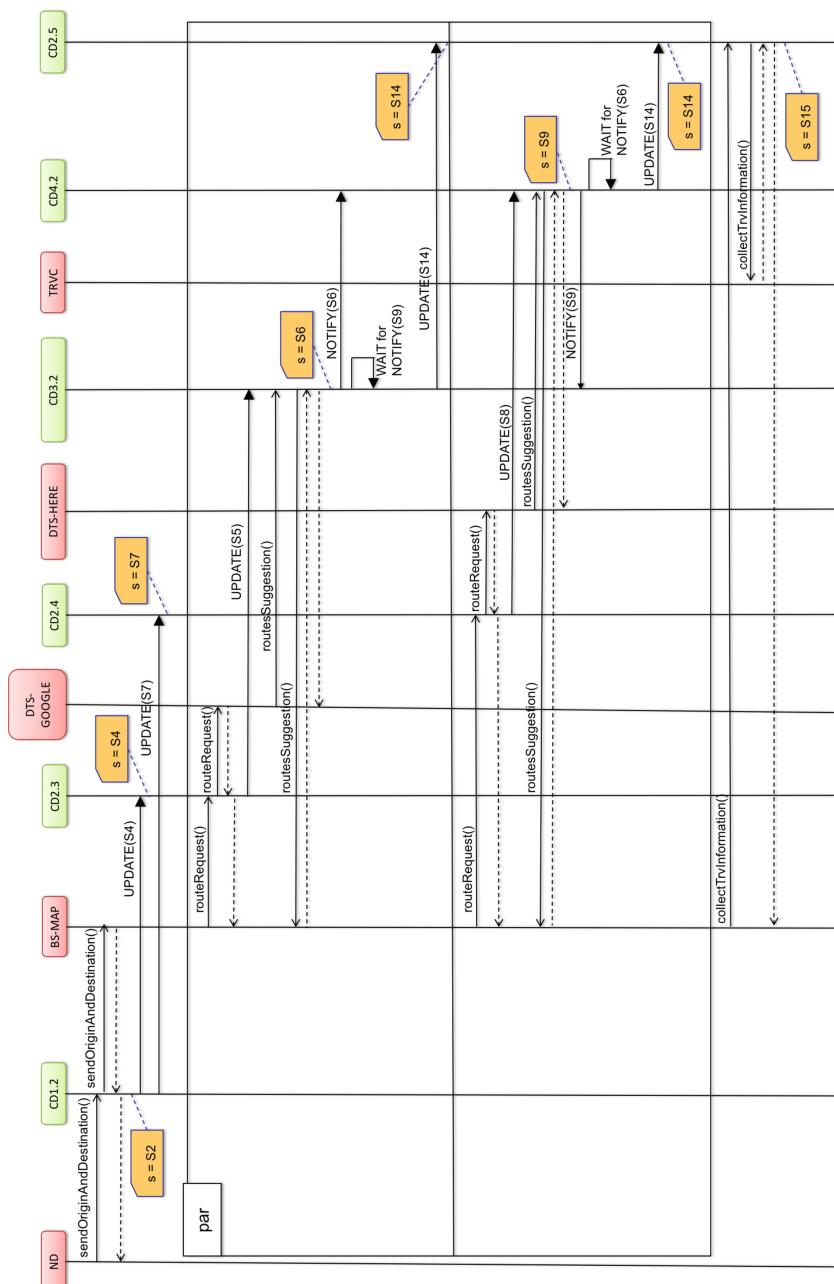


Fig. 12. Messages exchange for the running example

$CD_{i,j}$ sends an UPDATE message in order to inform, about the state change, only those CDs whose execution can progress from the new current global state. Thus, if a $CD_{h,k}$ receives a request to perform a task t while its local execution is in a state where t is not allowed, then it WAITS for receiving an UPDATE message on a state from which t is allowed.

Let us now consider the choreography specification enhanced with the introduction of variation points as done in Sect. 4. Conceptually, as far as context changes are concerned, beyond performing pure distributed coordination, CDs also manage Call Choreography tasks by activating/deactivating the related choreography variations according to the sensed choreography context. This calls for an enhanced notion of CM that contains dedicated tuples for expressing coordination logic variations, one for each Call Choreography variation. Upon reaching a variation point, and hence a Call Choreography task, the involved CD, among other actions, perform the following steps: (i) invokes the associated context evaluator; (ii) selects a suitable variation, according to the response of the context evaluator that characterizes the current choreography context; (iii) notifies the CD in charge of initiating it; and (iv) waits for being notified by the CD(s) in charge of terminating the variation execution. Note that context changes can happen at any point in time, but they are actually taken into account only within the scope of a Call Choreography. Furthermore, while executing a variation associated to a Call Choreography, further context changes are not considered until a new execution of the Call Choreography. As far as choreography specification changes are concerned, CD are external controllers that realize *multiple interacting feedback loops* enabling choreography evolution in response to possible changes applied to Call Choreographies. Moreover, following the philosophy of models at run time [1, 9, 11], CDs manage their own CMs as follows. A CM is an XML file codifying those coordination information that, being dynamically extracted from the current choreography specification, the CD has to locally known in order to suitably coordinate with the other CDs while enforcing the new global choreography resulting from both the modified Call Choreographies and the sensed context for them. It is worthwhile to mention that the overhead due to the context evaluation does not affect the “performance” of the running choreography. In fact, context evaluation is periodically performed by the run-time support offered by the execution environment using dedicated computational resources. This means that the computational resources used to execute the choreography are not affected by context evaluation.

Concretely, CDs implement a MAPE loop [18, 22], i.e., an abstraction of a feedback loop where the dynamic behavior of the managed system is controlled using an autonomic manager. Thus, by following the architectural blueprint for autonomic computing [19], CDs implement four phases: Monitor (M), Analyzer (A), Planner (P) and Executor (E). Moreover, in order to enable choreography evolution without incurring in disruptive interruptions, the CDs’ MAPE loop makes use of the notion of choreography *quiescent* state. For a formal definition of the seminal general notion of quiescent state we refer to [23]. In our context, a choreography is in a quiescent state with respect to given Call Choreography

changes if the CDs affected by the change are in a quiescent state. Roughly speaking, a CD is in a quiescent state if: (i) the portion of its CM, which is affected by the change, does not involve the current execution state, and (ii) it has completed all service-CD and CD-CD interactions required to perform a task and it has not yet started interactions required for a new task.

After the choreography modelers change the current choreography specification by modifying the set of variations associated to Call Choreographies, the synthesis processor re-synthesizes (only) the CMs that are affected by the change, and re-distribute them to the interested CDs. Moreover, the processor may also synthesize a new ChorSpec due to the required CDs addition and removal, or substitution of their CMs. Thus, in our context, changing the choreography specification and re-synthesizing the affected portions of already deployed artifacts (beyond possibly adding new ones and removing old ones), at run time, means dealing with the following cases:

- the CM of an already deployed CD is substituted when the interactions among the participants controlled by the CD have been modified due to, e.g., participant service(s) substitution, or specification of new interactions between the already deployed participant services;
- a new CD is added when a new participant has been introduced, or new interactions have been specified among existing participants not already controlled by any CD;
- a CD in between two participants is removed when at least one of them is not involved in the choreography anymore. As clarified below, CDs addition, removal, or substitution of their CMs are handled by specific adaptation rules and related mechanisms.

The CDs MAPE loop is then realized as follows:

1. **Monitors** – each interested monitor pre-processes the new CM and ChorSpec, gathers differences with the previous ones, and informs the analyzer.
2. **Analyzers** – by reasoning on the gathered differences, each interested analyzer establishes the nature of the change, i.e., CD addition or removal, or CM substitution.
3. **Planners** – each interested planner selects suitable actions to enable choreography evolution according to the supported adaptation rules, related mechanisms, and the results of the analyzer. Before executing the required adaptations, a choreography quiescent state has to be reached, and kept all throughout the adaptation process. For this purpose, each interested planners communicate each others to check if their CDs are all in a quiescent state. If it is the case, the planner activates the executor by providing it with the adaptation plan, e.g., substitution of the CM and, according to a new ChorSpec, substitution of one of the two controlled services. Contrariwise, the planner waits for the quiescent state to be reached (if possible). Note that the way CD planners are realized points out the distributed nature of the overall MAPE loop we realize by means of multiple interacting feedback loops.

4. **Executors** – after each interested executor is activated by its planner, the executor is in charge of keeping the quiescent state of its CD for the time that is needed to realize the received adaptation plan. For this purposes, the executor first informs the instance of the distributed coordination algorithm run by its CD to buffer possible incoming service requests. Secondly, the executor interacts with the EE to reconfigure the current architectural configuration, e.g., by deploying/undeploying services and (re-)establishing the new dependencies. Finally, pending service requests are handled once the adaptation process terminates and after the affected portion of the choreography is re-enacted.

6 Related Work

The work described in this chapter is related to several approaches developed for automated choreography realization and for adaptive systems engineering.

Automated Choreography Realization – The approach described in [16] enforces choreography’s realizability by automatically generating monitors. Each monitor acts as a local controller for its peer. This approach obtains monitors by iterating equivalence-checking steps between two centralized models of the whole system. It produces one of the models by composing the peer labeled transition systems (LTSs) assuming synchronous communication. It produces the other by composing the peer LTSs assuming asynchronous communication. The concept of monitor is similar to our Coordination Delegate (CD). However, our approach synthesizes CDs without producing a centralized model of the whole system, hence preventing state explosion. Furthermore, the approach in [16] is more theoretical and generates only the model of the monitors. In contrast, our approach synthesizes both the actual code implementing the CDs and their deployment schema.

The approach in [21] checks the conformance between the choreography specification and the composition of participant implementations. The described framework can model and analyze compositions in which the interactions can also be asynchronous and the messages can be stored in unbounded queues and reordered if needed. Following this line of research, the authors provided a hierarchy of realizability notions that forms the basis for a more flexible analysis regarding classic realizability checks [20, 21]. These two approaches are novel in that they characterize relevant properties to check a certain degree of realizability. However, they statically check realizability and do not automatically enforce it at run time.

The ASTRO toolset supports automated composition of Web services and the monitoring of their execution [31]. It aims to compose a service starting from a business requirement and the description of the protocols defining available external services. More specifically, a planner component automatically synthesizes the code of a centralized process that achieves the business requirement by interacting with the available external services. Unlike our approach, ASTRO

deals with centralized orchestration-based business processes rather than fully decentralized choreography-based ones.

The CIGAR (Concurrent and Interleaving Goal and Activity Recognition) framework aims for multigoal recognition [17]. CIGAR decomposes an observed sequence of multigoal activities into a set of action sequences, one for each goal, specifying whether a goal is active in a specific action. Although such goal decomposition somewhat recalls our choreography decentralization, goal recognition represents a fundamentally different problem regarding realizability enforcement. That is, goal recognition concerns learning a goal-based model of an agent by observing the agent's actions while interacting with the environment. In contrast, realizability enforcement produces a decentralized coordination logic out of a task-based specification of the choreography.

Given a set of candidate services offering the desired functionalities, the TCP-Compose* algorithm identifies the set of composite services that best fit the user-specified qualitative preferences over non functional attributes [28]. Our approach could exploit this research to extend the discovery process to enable more flexible selection of services from the registry.

Adaptive Systems Engineering – Several different research efforts in adaptive systems engineering exist, as well as comparative studies among different approaches [2] and rigorous characterizations of the key conditions that enable systems to adapt to changes in their environment or their requirements in order to continue to fulfil their goal [27].

The work in [6] describes a method to automatically synthesize *intelligent service proxies*. They enable self-adaptation of a service-based system via run-time selection of suitable participants in a workflow. Differently from our approach, the focus of this work is on the dynamic selection of the participant services.

In [14] an adaptation framework for self-healing systems is proposed, which consists of monitoring, analysis/resolution, and adaptation. In [7], this framework is integrated into a real-world software-intensive system which already features built-in adaptation mechanisms.

A declarative approach to model service orchestrations able to adapt to run-time changes is described in [10]. A specification language plus an ad-hoc engine able to interpret the specified models are described. The focus of this work is on orchestration that is a fundamentally different composition approach from choreography.

The work in [30] presents REFRACt, a self-adaptive framework for avoiding failures in configurable systems. REFRACt monitors the system for failures, and then triggers an algorithm to search for a passing reconfiguration, close to the original. It then updates a guard which prevents future clients from reconfiguring into known bad states. The idea of avoiding the execution of faulty code after a failure is observed consider adaptiveness from a different perspective with respect to the previously discussed approaches that are based on feedback control.

A formal approach that considers adaptiveness as continuous verification is discussed in [13]. The presented approach is applied on an example of choreography and is focused on adapting the system with respect to the assurance of reliability and performance properties.

The work in [15] describes a dynamic self-adaptation pattern for distributed transaction management in service-oriented applications. This pattern involves the coordination of distributed stateful transactions in which updates to more than one service need to be coordinated.

A goal-based service composition approach is presented in [29]. It leverages ontology reasoning and hierarchical task network planning in order to dynamically compose existing services out of an high-level specification of the user's goals.

7 Conclusions and Future Work

We have described a practical approach to the distributed enforcement of service choreographies, whose coordination logic is dynamically evolved in response to choreography specification changes and context changes. It takes as input a BPMN2 choreography specification and passing through an intermediate automata-based model, called Choreography-explicit Flow Model (CeFM), derives a set of Coordination Models (CMs), one for each participant service. Each CM contains coordination information that are used at run time by additional software entities, called Coordination Delegates (CDs). The CDs collaborate with each other to coordinate the global interaction of the services so to realize the specified choreography in a fully distributed way. Each CD runs its own instance of a distributed coordination algorithm and implements a MAPE loop. All together, CDs realize an overall MAPE loop by means multiple interacting feedback loops, hence enabling choreography evolution at the level of both the supervised participants (local evolution) and the emergent collaboration among them (global evolution).

We have illustrated the applicability of our method by means of a running example in the Intelligent Transportation Systems domain. The application to this example has shown that our approach is viable towards proposing a practical method that can be used by choreography developers.

The ongoing implementation of the described method is part of a model-based tool chain released under the FISSi initiative⁹ to support the development of evolving choreography-based systems in the CHOReVOLUTION EU H2020 project¹⁰, which is a follow-up of the CHOReOS EU FP7 project¹¹. Within CHOReVOLUTION we are enhancing the synthesis processor, namely CHOReOSynt, released by CHOReOS to deal with the dynamic evolution of choreographies. CHOReOSynt has already been applied to other real-world use cases in the *Airport*, and *Marketing and Sale* domains. The implementation of our method and of some simulated case studies, plus related documentation, are available at the CHOReOSynt web site: <http://choreos.disim.univaq.it>.

As future work, in order to achieve the even more ambitious objectives of the CHOReVOLUTION project and to improve the applicability of the approach,

⁹ http://www.ow2.org/view/Future_Internet.

¹⁰ <http://www.chorevolution.eu/>.

¹¹ <http://www.choreos.eu/>.

we plan to extend it to deal with protocol adaptation and non-functional requirements. As far as protocol adaption is concerned, our plan is to synthesize the adaptation logic required to adapt at run time the actual interaction protocol of those services that do not exactly fit the choreography roles. Concerning non-functional requirements, we will start by considering QoS degradation during the choreography execution, as services currently involved in the choreography no longer “perform” as expected. On the practical side, the source of this type of run-time changes can be, e.g., changing network conditions, degrading computational resources of the execution environments where the services are deployed, upgrading the version of the middleware on top of which the services run, and remote service substitution.

Acknowledgment. This research work has been supported by the Ministry of Education, Universities and Research, prot. 2012E47TM2 (project IDEAS - Integrated Design and Evolution of Adaptive Systems), by the European Union’s H2020 Programme under grant agreement number 644178 (project CHOReVOLUTION - Automated Synthesis of Dynamic and Secured Choreographies for the Future Internet), and by the Ministry of Economy and Finance, Cipe resolution n. 135/2012 (project INCIPCT - INnovating CIty Planning through Information and Communication Technologies).

References

1. Andersson, J., Baresi, L., Bencomo, N., de Lemos, R., Gorla, A., Inverardi, P., Vogel, T.: Software engineering processes for self-adaptive systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) *Software Engineering for Self-Adaptive Systems II*. LNCS, vol. 7475, pp. 51–75. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_3
2. Angelopoulos, K., Souza, V.E.S., Pimentel, J.: Requirements and architectural approaches to adaptive software systems: a comparative study. In: SEAMS 2013, pp. 23–32 (2013)
3. Autili, M., Inverardi, P., Tivoli, M.: Automated synthesis of service choreographies. *IEEE Softw.* **32**(1), 50–57 (2015)
4. Autili, M., Tivoli, M.: Distributed enforcement of service choreographies. In: FOCLASA (2014)
5. Brun, Y., et al.: Engineering self-adaptive systems through feedback loops. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) *Software Engineering for Self-Adaptive Systems*. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_3
6. Calinescu, R., Rafiq, Y.: Using intelligent proxies to develop self-adaptive service-based systems. In: TASE 2013, pp. 131–134 (2013)
7. Câmara, J., Correia, P., De Lemos, R., Garlan, D., Gomes, P., Schmerl, B., Ventura, R.: Evolving an adaptive industrial software system to use architecture-based self-adaptation. In: SEAMS 2013, pp. 13–22 (2013)
8. Cardellini, V., Casalicchio, E., Grassi, V., Iannucci, S., Lo Presti, F., Mirandola, R.: MOSES: a framework for QoS driven runtime adaptation of service-oriented systems. *IEEE Trans. Softw. Eng.* **38**(5), 1138–1159 (2012)

9. Cheng, B.H.C., et al.: Using models at runtime to address assurance for self-adaptive systems. In: Bencomo, N., France, R., Cheng, B.H.C., Afmann, U. (eds.) Models@run.time. LNCS, vol. 8378, pp. 101–136. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08915-7_4
10. Cugola, G., Ghezzi, C., Pinto, L.: DSOL: a declarative approach to self-adaptive service orchestrations. Computing **94**(7), 579–617 (2012)
11. de Lemos, R., et al.: Software engineering for self-adaptive systems: a second research roadmap. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 1–32. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_1
12. European Commission: Digital Agenda for Europe - Future Internet Research and Experimentation (FIRE) initiative (2015)
13. Filieri, A., Ghezzi, C., Tamburrelli, G.: A formal approach to adaptive software: continuous assurance of non-functional requirements. Formal Aspects Comput. **24**(2), 163–186 (2012)
14. Garlan, D., Schmerl, B.: Model-based adaptation for self-healing systems. In: Proceedings of WOSS 2002, pp. 27–32 (2002)
15. Gomaa, H., Hashimoto, K.: Dynamic self-adaptation for distributed service-oriented transactions. In: SEAMS 2012, pp. 11–20 (2012)
16. Güdemann, M., Salaün, G., Ouederni, M.: Counterexample guided synthesis of monitors for realizability enforcement. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, pp. 238–253. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33386-6_20
17. Hu, D.H., Yang, Q.: CIGAR: concurrent and interleaving goal and activity recognition. In: AAAI 2008, pp. 1363–1368 (2008)
18. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing - degrees, models, and applications. ACM Comput. Surv. **40**(3), 1–28 (2008)
19. IBM: An Architectural Blueprint for Autonomic Computing. White Paper, 4th edn., IBM (2006)
20. Kazhamiakin, R., Pistore, M.: Analysis of realizability conditions for web service choreographies. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 61–76. Springer, Heidelberg (2006). https://doi.org/10.1007/11888116_5
21. Kazhamiakin, R., Pistore, M.: Choreography conformance analysis: asynchronous communications and information alignment. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 227–241. Springer, Heidelberg (2006). https://doi.org/10.1007/11841197_15
22. Kephart, J., Chess, D.: The vision of autonomic computing. IEEE Comput. **36**(1), 41–50 (2003)
23. Kramer, J., Magee, J.: The evolving philosophers problem: dynamic change management. IEEE Trans. Softw. Eng. **16**(11), 1293–1306 (1990)
24. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: 2007 Future of Software Engineering, FOSE 2007, pp. 259–268, May 2007
25. Lamport, L.: Time clocks, and the ordering of events in a distributed system. Commun. ACM **21**, 558–565 (1978)
26. Leite, L., Moreira, C.E., Cordeiro, D., Gerosa, M.A., Kon, F.: Deploying large-scale service compositions on the cloud with the CHORéOS Enactment Engine. In: Proceedings of 13th IEEE International Symposium on Network Computing and Applications (NCA 2014), pp. 121–128. IEEE (2014)
27. Salifu, M., Yu, Y., Bandara, A.K., Nuseibeh, B.: Analysing monitoring and switching problems for adaptive systems. J. Syst. Softw. **85**(12), 2829–2839 (2012)

28. Santhanam, G.R., Basu, S., Honavar, V.: TCP – compose^{*} – a TCP-net based algorithm for efficient composition of web services using qualitative preferences. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 453–467. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89652-4_34
29. Song, S., Lee, S.-W.: A goal-driven approach for adaptive service composition using planning. *Math. Comput. Model.* **58**(1–2), 261–273 (2013)
30. Swanson, J., Cohen, M.B., Dwyer, M.B., Garvin, B.J., Firestone, J.: Beyond the rainbow: self-adaptive failure avoidance in configurable systems. In: FSE 2014, pp. 377–388 (2014)
31. Trainotti, M., et al.: ASTRO: supporting composition and execution of web services. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 495–501. Springer, Heidelberg (2005). https://doi.org/10.1007/11596141_39

Models for the Consistent Interaction of Adaptations in Self-Adaptive Systems

Nicolás Cardozo¹(✉), Kim Mens², and Siobhán Clarke¹

¹ Future Cities, DSG, Trinity College Dublin, College Green 2, Dublin 2, Ireland
✉ {cardozon,Siobhan.Clarke}@scss.tcd.ie

² ICTEAM Institute, Université catholique de Louvain, Place Sainte-Barbe 2,
1348 Louvain-la-Neuve, Belgium
kim.mens@uclouvain.be

Abstract. Self-adaptive systems enable the run-time modification, or dynamic adaptation, of a software system in order to offer the most appropriate behavior of the system according to its context of execution and the situations of its surrounding environment. Depending on the situations currently at hand, multiple and varied adaptations may affect the original behavior of a software system simultaneously. This may lead to accidental behavioral inconsistencies if not all possible interactions with other adaptations were anticipated. The behavioral inconsistencies problem becomes even more acute if adaptations are unknown beforehand, for example, when new adaptations are incorporated to the system on the fly. Self-adaptive systems must therefore provide a means to arbitrate interactions between adaptions at run time, to ensure that there will be no inconsistencies in the system's behavior as adaptations are dynamically composed into or withdrawn from the system. This chapter presents existing approaches that allow the development of self-adaptive systems and management of the behavioral inconsistencies that may appear due to the interaction of adaptations at run time. The approaches are classified into four categories: formal, architectural modeling, rule-based, and transition system approaches. Each of these approaches is evaluated with respect to the assurances they provide for the run-time consistency of the system, in the light of dynamic behavior adaptations.

1 Introduction

Self-adaptive systems enable the dynamic adaptation of software systems with respect to their internal state, and the situations of their surrounding execution environment. Such information about a system is gathered respectively using internal system monitors and sensor networks. Dynamic software adaptation of a system takes place with the objective of providing a better Quality of Service (QoS), or the most appropriate service behavior for the system with respect to the current situations of its execution environment [29].

System monitors and sensors gather information about the various internal and external situations to which software systems could respectively adapt. Software systems' adaptation based on such information posses two requirements on

the systems design. Systems must: (1) provide a continuous adaptation process, where components can be introduced into, or removed from the system unannounced without stopping the system's execution, and (2) assure that the system's adaptations remain consistent with respect to the expected behavior of the system and other adaptations made to the system [43]. The former characteristic is important to ensure that systems effectively change its structure or behavior, to provide a more appropriate service with respect to its changing environment. The latter characteristic is important to avoid errors or system crashes that may arise due to the (incorrect) interaction between adaptations during the execution of the system.

There are different techniques fostering dynamic adaptation of software systems with respect to their execution environment, for example, by using dynamic software recompositions or program upgrades. Ensuring the consistency of these adaptations is, however, a more challenging task requiring that the system behaves as expected in all possible situations in which it may be adapted — that is, all possible combinations of adaptations are consistent. The number of possible adaptation combinations increases exponentially with the definition of every new adaptation. It is thus unfeasible to design a model that takes into account all combinations beforehand.

Self-adaptive systems are extended with orchestration models to deal with the complexity of adaptations and their interaction. We focus the discussion of this chapter on the description of different approaches for self-adaptation that enable the reasoning and managing of adaptations' properties. In particular, the approaches presented hereinafter are assessed with respect to the consistency assurances they provide for adaptations and their run-time interaction. Rather than presenting concrete self-adaptive development models for consistency assurance, we describe the general approaches that can be (and are) used to provide a concrete orchestration model. To structure the discussion and assessment of the approaches, we group them into four categories, as follows: (1) formal approaches, backed by a mathematical formalism in which consistency of the system can be verified; (2) architectural modeling approaches, which are based on abstract architectural descriptions (*e.g.*, UML models) of the system structure to represent and manage adaptations; (3) rule-based approaches, which use production rule systems to describe adaptations and actively verify the validity of their rules; and (4) transition system approaches, which are defined in a state-transition fashion describing adaptations and managing the actions between them.

2 Background

Before delving into the analysis of self-adaptive system and their consistency assurance models, we describe the properties that these systems should satisfy with respect to their expected execution environments. Self-adaptation is envisioned to provide the most benefits for large-scale, critical, time-constrained, or distributed systems. The three properties presented here are extracted taking these application domains into account [14]. An application domain of adaptive

systems is that of Cyber-Physical Systems (CPSs), for example, smart-home systems, where different household appliances can adapt their behavior according to user preferences, interaction with other appliances, their internal state, or conditions from the surrounding environment. The smart-home system is used as a unifying example throughout our analysis of existing consistency models for self-adaptive systems.

2.1 Case Study: Smart-Home Systems

Smart-home systems have become a *de-facto* example to highly dynamic systems exhibiting adaptations with respect to their surrounding environment [20, 30]. In smart-home systems, household appliances (*e.g.*, stereo, television, windows, heater, lights, ...) are interconnected creating a (wireless) sensor network. Different services can be defined on top of the sensor network to control devices remotely or even autonomously, for example, remote light switching or automatic temperature adjustment. Household inhabitants have different preferences about the settings and usage of appliances. Hence, services in a smart-home system would benefit from dynamic customization, taking into account users' preferences and the surrounding execution environment of the service.

As a concrete example of adaptation and interaction we will use the case of two services deployed in the system: user detection, and room ambient services. The user detection service is used to identify users moving about a smart-home. User presence can be detected using infrared or RFID sensors, their identity can be verified using their mobile devices (through the wifi connection), RFID tags (through RFID readers), or voice and facial recognition (respectively using noise sensors and cameras deployed around the house). The room ambient services use ambient sensors such as temperature, humidity, air quality, noise, or photocells to adjust room conditions to users' preferences. Conditions in a room change autonomously as users are identified in a room.

Adaptation scenario: Having acquired a user's identity, appliances can adapt to provide more appropriate room ambient conditions, taking into account both, users' preferences and the external situations of the environment. For example, room temperature can be set based on a user's particular comfort level, current weather conditions, and season of the year. The way in which the system satisfies these situations leads to many different possibilities of adaptation. For example, opening the windows to cool the room down if it is neither raining nor winter, or turning on the air-conditioning if the user would rather keep the windows closed. The user identification service is also responsible for contacting the authorities in case of emergencies. This process may vary according to the particular emergency. For example, if users cannot be identified properly, an alert is sent to the police and the home is locked to avoid burglary. In case of fire or flood, however, the fire department should be contacted instead, giving free access into the house for firefighters. If identified users are detected to have a (medical) emergency (*e.g.*, falls, or severe injuries) the appropriate medical team is contacted and granted access into the home.

In the presented scenario of the smart-home system example, it is possible to see that interaction is a central factor to dynamic adaptations. Moreover, adaptation interaction is inherent to services sharing a base application domain. Therefore, interactions must always be accounted for, otherwise the system may present inconsistent or erroneous behavior. In our example, inconsistencies due to interactions can occur whenever both the user detection and room ambient services are required to contact the authorities in customized ways. The system needs to ensure the contacted authorities are consistent with the detected emergency, the firefighters should be called with the appropriate information to be prepared for the situation (*i.e.*, fire or flood). Additionally, both services actuate over the same appliances (*e.g.*, doors and windows) depending on the situation at hand. In case of burglary, the user detection service may intend to close the windows, while the room ambient service may require to keep them open to maintain the room's temperature.

Therefore, we assert self-adaptive systems require a model that effectively expresses and manages interaction between adaptations in order to ensure the consistency of the system. To accomplish this, the model must manage *all* possible interactions between adaptations, even if not explicitly defined beforehand.

2.2 Consistency Assurance Requirements

Software models for the development of self-adaptive systems should satisfy the following three properties in order to assure the consistent interaction between adaptations.

M.1 Abstraction: Adaptations are the central entity to model self-adaptive systems. Hence, models managing these systems must provide the capability to represent (at run time) the system as a composition of its adaptations, their defined interactions (if any), and their states. The purpose of these representations is to support the run-time verification of system properties in order to ensure their consistency. The system representation at run time must be concise and express the different states of the system and the transitions (actions) between such states. An additional restriction is set for the abstraction property. Abstractions to manage the consistency between adaptations should correspond, as much as possible to the abstractions to develop the system. This is to avoid inconsistencies between the verification and execution layers of the system.

M.2 Run-time Consistency: Interactions between adaptations may be unknown as adaptations are introduced to the running system. New or unknown adaptations can be harmful to the system as their behavior may conflict with other adaptations already existing in the running system, for example, by providing a contradictory behavior. This can be evidenced in the smart-home system with the introduction of the room ambient adaptation that opens a room's windows. This new behavior contradicts that of the user detection service in case of burglary. Every time an adaptation is introduced to the running system, it should be guaranteed that this does not lead to

inconsistent system states. At run time, adaptations should be verified to assure their interactions are consistent.

M.3 Offline Consistency: Run-time consistency verification can be too heavy-weight, specially in large systems. To mitigate this, the model should also provide a capability to reason about system properties, that may influence its overall run-time consistency, as interactions between adaptations are defined. The analysis process must also be extensible to take into account interactions that may occur between adaptations at run time, even if they were not explicitly defined.

3 Approaches for Consistency Management

We now provide an overview of the different models that can be used to manage adaptation interaction in self-adaptive systems. As previously mentioned, studied approaches are characterized in four main groups: *formal*, *architectural modeling*, *rule-based*, and *transition system*. Each of the approaches is presented in a structured way where we describe the generalities of the model, how it enables consistency management capabilities, and an evaluation of the three consistency assurance requirements. The discussion and evaluation of the approaches is driven via the smart-home system example introduced in Sect. 2.1. A comparative analysis of the approaches is presented in Sect. 4.

The four classification groups are taken from the commonalities of concrete run-time models used for the coordination of (self-adaptive) software systems. Therefore, to complement the description of the different approaches, we present a description of a concrete run-time model based on the approach managing the run-time adaptations. Note that by run-time models, here and in the remainder of the chapter, we mean an abstraction of the system's state space and the different actions between states that can be accessed and maintained dynamically.

3.1 Formal Approaches

Formal approaches are the basis for the specification and analysis of software systems. These approaches are normally used at early stages of the development process, benefiting from the provided mathematical tools to prove or identify interesting system properties. Three common uses of formal approaches in software systems development include: (1) disambiguate the concepts embedded in the system, (2) analyze, verify, or prove properties about the system, and (3) raise the level of abstraction of the system. While formal approaches are normally developed in parallel to an adaptive system (and therefore do not provide adaptation abstractions), we include them in the our study due to their powerful reasoning capabilities and their tight integration with the development of self-adaptive systems. Here, we focus on the latter two categories, paying special attention to the approaches that can be used in the development of dynamic adaptations. Moreover, we highlight the approaches that focus on the verification of interaction consistency between adaptations. Formal approaches are

normally developed as abstract reasoning engines that are used to specify a given (adaptive) system, or are coupled to one as a reasoning engine. Therefore, the approaches presented hereinafter may not be directly applied to develop self-adaptive systems, but could be used together with the other approaches described in later sections to develop the system and assure its consistency.

The formal approaches described here are: *computational logic*, *algebraic specification*, and *Satisfiability Modulo Theories*, selected for our analysis as they have been used in the implementation of concrete models for the assurance of consistency of self-adaptive systems.

Computational Logic. Computational logic is used to reason about computations of software systems through logic formalisms. The logic programming paradigm embodies computational logic by putting forward mathematical logic statements as the base instructions to build a software system. Statements are used to express properties about the states in a system, while fact rules express transitions between these states. In order to analyze a system, its states are given an abstract representation appropriate for the intended purpose of the system. For example, using an Abstract Syntax Tree (AST) representation, or control flow graph to reason about the consistency of the system's evolution, querying the system to identify behavioral or design errors [19]. Different logic systems can be used to further reason about the consistency of the system. For example using the Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) temporal modal logics [8], it is possible to reason about properties that should be satisfied over a period of time (*e.g.*, point liveness) [68].

Logic systems can be used for the specification of dynamically adaptive systems by associating *statements*, *transition rules*, and *queries* with adaptable program elements. Statements express the conditions in which an adaptation is applicable. For example, in our case study, the statements `:when (livingroom_sensor == "sunny") or window(livingRoom, sunny, open)` express the conditions in which windows open in the room ambient service. These statements are applied to the windows if two conditions are true, the current room is the living room, and the weather sensors signals it is sunny. Transition rules verify statements' validity, and based on the result adaptations are composed into or withdrawn from the system. If the statements match the conditions specified in the transition rule, then the system transitions to that state, and the adaptation associated with the statement (`open` in our case) is incorporated with the system. The transition rule matching the window statement as specified in rules in a rule-based system is shown in Line 5 of Sect. 3.3. Finally, a set of queries (or formulae) may be used to define the properties that should hold during the execution of the system, for example, a modal formula query, $\square(\text{window} == \text{"closed"} \wedge \text{weather} == \text{sunny})$, verifies that the windows are always closed, unless it is sunny.

Interaction inconsistencies between adaptations can be identified using logic programming, for example, if at least one of the defined queries is unsatisfied. To assure the consistency of the system two possibilities exist. Statically, all possible

combinations between adaptations would have to be tested running all applicable queries for each scenario. Dynamically, whenever an adaptation takes place, all queries associated with the adaptations currently composed in the system are verified. While the two possibilities are plausible, note that both require the complete verification of all defined queries. This can be too heavyweight for large systems with many queries. Moreover, an additional abstract model of the system is used to verify the system properties, where, *a priori*, the states of the system and transitions between such states are represented.

Consistency requirements satisfaction: Computational logic provides a model to reason about the system in which system states are defined as statements, transition between states are defined as transition rules, and management of adaptations takes place via querying the system states. Consistent interactions between adaptations in self-adaptive systems defined using logic programming, as presented here, are given by sets of adaptations that can satisfy all queries associated with them, satisfying the run-time consistency requirement. Additionally, analysis of interactions and correctness of the logic system can be performed statically, satisfying the offline consistency requirement. This last requirement is further supported by the use of other logic systems (complemented by system's abstractions *e.g.*, AST) to reason about properties of the system. The use of computational logic enable the adaptation (via parametric adaptation) via the model verification of statement and transition rules, however, in existing approaches adaptations cannot be explicitly defined as system elements during the development. Hence, the abstraction requirement is dissatisfied.

Application to self-adaptive systems: Logic approaches have been used mainly for the specification of adaptive systems, applied to the specification of sockets [68], or monitoring in embedded systems [64]. Computational logic approaches can also be used to manage self-adaptive systems if they are extended to express adaptations and interactions between program entities (similarly to the extensions required to reason over source-code). Since the systems must be extended and modified, the verification and reasoning techniques defined with computational logic must be revisited. While logic programming languages cannot be used directly in the implementation of full-fledged self-adaptive systems, the ideas can serve as the basis to develop self-adaptive software systems as it is the case in rule-based approaches (*cf.* Sect. 3.3).

Algebraic Specification. Different mathematical formalisms can also be used to model and foster formal reasoning of software systems and their properties. The idea behind these approaches is to propose abstractions for a system, within the boundaries of a specific mathematical formalism in such a way that they can be used as decision engines for different system properties. Here we present those formalisms suitable to reason about self-adaptive systems.

Process algebras [24,34] are used to model concurrent processes, providing high-level abstractions for operations between processes such as parallel composition, communication, replication, restriction, and synchronization. The most

prominent representative of process algebras is the π -calculus specification [48], in which it is possible to fully express the way in which different components communicate. In particular, it is possible to express the conditions under which two processes communicate (via restrictions). Restrictions can be used to limit how and when the main process of the system communicates. Similarly, restrictions can capture the context (*i.e.*, situations in the surrounding environment) in which processes execute. Such contexts can be used to define the different adaptations of a system such that composing different available processes will exhibit different behavior for different situations of their surrounding environment. In our case study's example, the restriction to open a room's window is that the weather is sunny, if this restriction $\nu l := \text{WEATHER} == \text{sunny}$, is satisfied, then the window state carried through process p will change from its current value \bar{s} to OPEN . Otherwise the adaptation is not composed in the system as the state remains the same S . The complete formula can be written as $\{\} \vdash \text{WINDOW} \xrightarrow{p} \{\text{state}\} \vdash S | (\nu l)(\bar{s} \rightarrow \text{OPEN})$.

Coalgebras, and in particular coalgebraic specification [38], have been used to express the dynamic behavior of systems. Coalgebraic specifications are structured using notions of inheritance and aggregation from Object-Oriented Programming (OOP), which are used to model state-based systems (where the state is considered as a black box). Dynamic adaptations can be expressed using coalgebraic specification by describing the conditions in which particular (pieces of) behavior should take place. A coalgebraic specification consists of a list of operations on the elements of the systems (*i.e.*, types of objects) and a list of assertions that such elements must satisfy.

These formalisms are used on their own as an abstract model to prove consistency or decidability properties about the systems they model. Process algebras enable developers to specify and restrict the computation flow of a system, which is used to reason about possible interactions between system entities. Inconsistencies in such interactions can be detected, for example, by identifying livelocks or deadlocks. Redundancies can be identified via behavioral equivalences (using bisimulation equivalence) of composed computation flows. Coalgebraic specifications allow us to reason about dynamic behavior of a system in terms of invariance and bisimilarity. Additionally, modal operators may be used within the coalgebraic specifications as invariants for reasoning about future states, and safety of progress/modal formulae among other system properties.

Consistency requirements satisfaction: Algebraic specification approaches are used to represent program properties, rather than to represent the actual program, and are consequently not used at run time. Therefore, it is not possible to have a run-time verification of interactions between program entities, dissatisfying the run-time consistency requirement. Similar to computational logic approaches, algebraic specification does not provide the concept of adaptation. Rather, adaptations are created implicitly, for example, using dedicated communication channels and restrictions. This complicates the definition of the system, as one of its concepts is hidden in the specification. Hence, these approaches also fail to satisfy the abstraction requirement. Note, however, that it is possible to capture the context

of the system. Using this property and the natural specification available to reason about different system properties the offline consistency requirement is satisfied.

Application to self-adaptive systems: In order to build self-adaptive systems using algebraic specification approaches, these need to be complemented with additional programming abstractions to represent and execute states of programs and the transitions between such states. Concrete models complementing algebraic specifications include abstract state machines [9], algebraic Petri nets [25], or alternating automata [61]. With the help of such models it is possible to introduce explicit notions of adaptations and structure a system's behavior in accordance to its algebraic specification.

Satisfiability Modulo Theories. Satisfiability Modulo Theories (SMT) are used as an approach to decide whether a set of first-order logic formulae describing a problem are satisfied. Here we present two approaches used in self-adaptive systems development.

Model Checking: Model checking is an analysis and verification technique based on the specification and restriction of system properties by means of logic formulae. Model checking techniques for the verification of logic formulae include abstract interpretation, partial order reduction, or automated theorem proving. The most prominent model checking technique is based on the boolean SATisfiability problem (SAT), which is commonly used to decide whether a system satisfies a set of logic formulae [53].

Model checking, and in particular SAT solvers, can be used in the context of self-adaptive systems as a reasoning tool. Interactions between adaptations within a system are usually represented by means of transition systems. Such representation can be used to create a model representation of the system using a specification language (*e.g.*, Promela) which can then be translated to logic formulae and verified by a model checking engine (*e.g.*, SPIN). Snippet 1 shows the specification of the events and adaptations of the smart-home system example using Promela. The transition rules describe the channel in which events are receive (*e.g.*, from sensors as `channel ? WeatherSensorEvent`). Processing of received information in the channel leads to the verification of the conditions in which one adaptation should be applied (*e.g.*, remove the `closeWindow` adaptation to introduce the `openWindow` one, whenever is sunny). Snippet 2 shows the translation of Promela rules into LTL formulae (the first two lines in Snippet 2) in SPIN, where [] and U are, respectively, the global and until modal operators.

Snippet 1. Promela specification

```

do
:: channel ? WeatherSensorEvent ->
if
:: (weather == Sunny) ->
atomic {!closeWindow; openWindow}
:: (season == Winter) ->
atomic {closeWindow}
fi
od

```

Inconsistencies between adaptations are identified if the model checker cannot assigned values to all variables in the formulae such that these are satisfiable (*i.e.*, their valuation is true).

Similar to algebraic specification, model checking is a verification approach, complementing the development and execution of self-adaptive systems. On its own, model checking does not provide means to explicitly represent adaptations, but rather it provides a way to verify consistency of transition rules defined between them. The abstraction requirement is thus not satisfied. Model checking does provide support for run-time verification from the implementation of the system by means of abstraction and symbolic algorithms built in SAT solvers, that could even be used at runtime [21,57], satisfying the run-time consistency requirement. Finally, model checking techniques enable the analysis of system properties when combined with a self-adaptive development approach. As an example, the EventCJ language provides high-level abstractions for the definition of adaptations and their dynamic behavior, which has been used to develop routing, mobile, and web systems. Such abstractions are specified using Promela and verified at development time to assure the validity of transition rules between adaptations in SPIN. However useful, we note that using multiple specifications of the system (*i.e.*, the state space, its specification model, and the program itself) can be error prone, as all representations must co-evolve over time. A unified model for the system and its specification is therefore desired. Given these properties, model checking satisfies the offline consistency requirement.

Constraint Satisfaction Problem: The Constraint Satisfaction Problem (CSP) is a technique to solve SMT problems by modeling the interactions between domain variables as restrictions in the system's functionality (*i.e.*, constraints). CSPs are specified by a set of variables $\mathbf{x} := (x_1, \dots, x_n)$ with domains $\mathbf{X} := (X_1, \dots, X_n)$, over which constraints are defined as clauses of the form $(\varPhi_1(\mathbf{x}), \dots, \varPhi_m(\mathbf{x}))$. Variables are associated with entities in the system (*e.g.*, represent system variables or objects), while clauses define relations between such variables. Similar to model checkers, CSP solvers find values for the variables in \mathbf{x} such that the valuation for all clauses $\varPhi_i(\mathbf{x})$ is true. If not all constraints are satisfied, the solver responds an “unsatisfiable” message, such message is possibly accompanied with the sets of constraints that could not be satisfied [10]. Using constraint solvers, adaptations can be modeled by defining constraints that associate structural and behavioral system properties with particular situations of its surrounding environment. Enabling adaptation management can be achieved using one of two approaches.

Snippet 2. LTL SPIN specification

```

[](closeWindow U(weather == Sunny))
[](openWindow U(season == Winter))
[](knownUser V window == closed)

```

A first approach [59], considers the adaptation of system's entities to maintain the satisfiability of the constraints defined in the system. Upon changes in the system's surrounding environment, system entities are adapted so that all constraints are satisfied. Consider the smart-home system in which constraints are extracted from the system's model, based on the current values of its variables as shown in Fig. 1. The model defines weak constraints, which are a special type of constraints that are not required to be satisfied if not all constraints can be satisfiable. Evaluation of the constraints in the righthand side of Fig. 1 results in an unsatisfiable set containing constraints {4., 5., 7.}, evidencing three inconsistencies in the system's execution. As a consequence, the system must be adapted to resolve as many inconsistencies as it is possible. A possible adaptation of the model to resolve such conflict would be change the state of the variable `Window.open` from `false` to `true` (*i.e.*, opening the window). Applying this resolution constraint 5. is the only constraint that remains in the unsatisfiable set. Note that this constraint cannot be resolved as possible adaptations will raise inconsistencies with other interacting constraints (*i.e.*, {6., 7., 4.}).

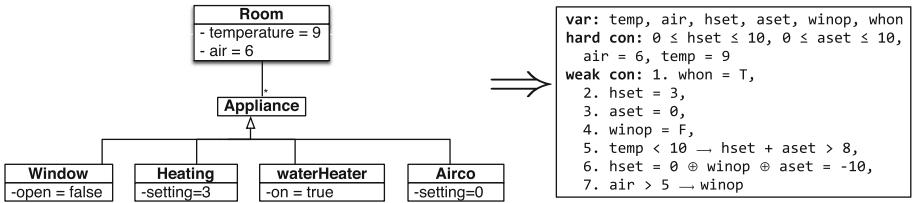


Fig. 1. Constraints associated with the smart-home system example (adapted from [59]).

A second approach, consists of using constraints to drive the system adaptation. In this approach entities are dynamically composed as part of the system if and only if their associated constraints are satisfiable in the context of the system's surrounding environment. Here, system adaptations are driven by changes in the surrounding environment—that is, introduction or withdrawal of data gathered by sensors. Such data can be explicitly represented by the definition of a context model [21]. Constraints are introduced to complement the context model, defining consistency relations between contexts. As new information is gathered, defined constraints in the system are evaluated. The system will then execute the actions associated with those contexts that do not lead to inconsistencies—that is, set of constraints with true valuations.

Constraint solvers are used to provide a decision layer for software systems by introducing constraints which define/restrict interactions between their entities. Constraint solvers do not provide an explicit abstraction to represent adaptations, but rather rely on the abstractions provided by the underlying system. Constraint solvers, thus, do not satisfy the abstraction requirement. Using

constraint solvers, adaptations take place after all constraints defined in the system are evaluated upon changes in the system's surrounding environment. Such changes can be consequences of conditions on the constraint variables' values changing, or constraints being added to or withdrawn from the system. The evaluation process (*i.e.*, constraint solving) ensures that as many constraints as possible are satisfied in the system (by means of adaptation). For example, the sm@rt framework [59] defines constraints extracted from the system's structure and adapts its values to maximize the number of satisfied constraints, and consequently the assure interactions between system entities are consistent. The framework has been used for the development of smart CPSs, and cloud infrastructure management. The constraint solving process, thus, satisfies the run-time consistency requirement. We note that while it is possible to evaluate the constraints at earlier stages of the development process, such evaluation is equivalent to run-time constraint evaluation. Since constraint solvers do not provide additional mechanisms to reason about the system's consistency beforehand, the decision requirement is not satisfied.

3.2 Architectural Modeling Approaches

Software systems' models as understood within the Model-Driven Engineering (MDE) community are a series of artifacts used to describe and design (parts of) a system. A large body of research has been applied to ensure the consistency among different models describing software systems as they evolve over time [52, 60]. Modeling approaches were initially thought out in the setting of static modification of the system's structure, where the system could be stopped and modified offline. Interest in the application of model modification at run time has given rise to the Models@runtime [49] community. Model modification at run time entails the interaction of three model component definitions of a system: (1) a model describing the main structure and adaptation points of the system, (2) a model specifying the system environment and variables relevant for the adaptation, and (3) the model's adaptations—that is, specific artifacts modifying the structure of the system with respect to its environment. These model components are overseen by a reasoning component, responsible for ensuring that adaptations are composed with the main structure of the system whenever the conditions specified by the system environment model are satisfied.

The architectural modeling approaches described here: *model transformations*, and *feature-oriented domain analysis* are selected as they have been used in the implementation of concrete models for the assurance of consistency of self-adaptive systems.

Model Transformations. Model transformation consists of a description of the system using a modeling language (like UML), where system adaptations are described as independent models. Changes signaled from the system are enacted by transitioning from the current model to a new model containing the required adaptations. A common approach to transition between models is to

define adaptations as graph transformation rules transitioning from one model to a model containing the adaptation [23, 56].

We follow the approach of Degrandart et al. [23] as an example to drive the discussion. The surrounding environment is defined using contexts domains, which are multi-dimensional spaces representing the domains of the situations that are relevant in the surrounding execution environment of the system. Specific situations in the environment are realizations of context domains, giving concrete values for (some of) its dimensions. The impact of a context change from context c to context d is specified by a transformation rule $r_{c,d}$ explicitly expressing the effect of context changes over model artifacts. In the smart-home system example take the base application (*i.e.*, without adaptations) as a context c , and the situation where a user is identified in a room as a context d . The effect on the class `Window` of changing from the former context to the latter one, is to add two new pieces of behavior to it; the functionality to `open()` and `close()` the windows, as shown at the top of Fig. 2. The example in Fig. 2 shows the adaptation of the model (for the aforementioned situation) matching the adaptation elements with the models to which they are applied. Given a model M_c of the system matching (through function m) with adaptation component L , M_c is transformed into a new model M_d matching (through function m') adaptation component R , by following the transformation rule $r_{c,d}$. The rule is applied whenever there is a context change from c to d —that is, the values defined in context c changed to give rise to context d on a given context domain. Note that not all possible transformation rules have to be defined. The model allows the transitive composition of transformation rules by relying on the sequential composition of graph transformations available in its execution language AGG [63].

Graph transformation approaches can also be considered as rule-based approaches, as they transition between system states via transitions rules, however, we classify them independently here, as the focus of this section is on the transformation of graph artifacts. Rule-based system are discussed in Sect. 3.3.

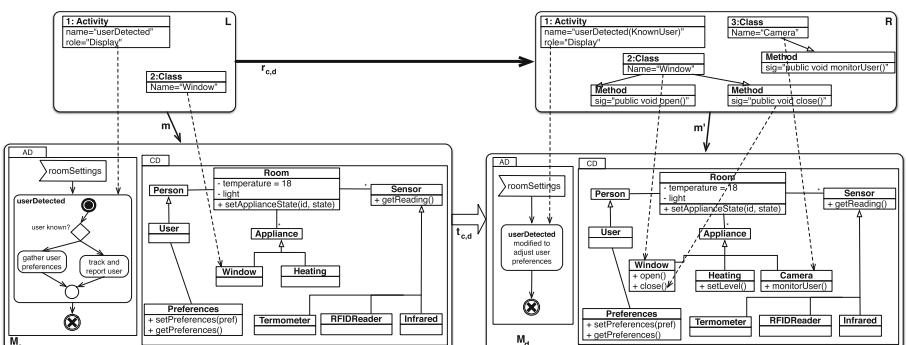


Fig. 2. Model transformation responding to context change, from no context active, to a user detected context in the smart-home system example.

There are two situations in which the interaction between adaptations can conflict: whenever one transformation rule deletes a model element required by a second transformation rule as an input, or whenever a transformation rule changes attribute values of the adaptation vector, such that a second transformation rule is no longer applicable. These conflicts are identified in the model by analyzing the coverability property of adaptations defined in the system —that is, analyzing if all adaptations defined in the system can be reached from the initial state of the system by means of graph transformation rules. The analysis of conflicts between adaptations takes place during the design phase of the system throughout a pair-wise analysis. Conflicts have to be resolved manually by domain experts, and re-analyzed. Once all conflicts are resolved, the application and its adaptations can be deployed.

Requirements satisfaction: Model transformation systems define adaptations as components in transformation rules. Each component contains the adapted object and is associated with the transformations the model object must undergo. The abstraction requirement is satisfied in these models as adaptations are explicitly represented by system entities. Moreover, the state of the system can be represented by the sequential composition of graph transformation from the base model. These models also provide the possibility to manage interactions between adaptations at design time, by means of the conflict identification analysis, thus, satisfying the offline consistency requirement. The run-time consistency requirement, however, is not satisfied in these models as conflicts are not verified at run time. If conflicts are overlooked during the design of the system, introduction or composition of adaptations could still yield an inconsistent behavior of the system due to conflicting adaptations.

Application to self-adaptive systems: Model transformations are fit to software systems that undertake a modeling approach for their conception. Introduction of a model transformation approach can be naturally integrated with the development process of the system, as it has been the case for smart visit applications [23]. Note that this model (and similar transformation approaches as Atlas Transformation Language (ATL) [41]) present disadvantages discouraging their use for the development of self-adaptive systems. First, the system is not represented by a single model, but by a set of models (*i.e.*, the basic representation, the activity model, and the adaptations of the system). Such proliferation of models can become complex to manage as systems grow and more entities and situations to which these can adapt are added. Second, interaction between adaptation dimensions (or their values) cannot be expressed. As a matter of fact, overlap between adaptation dimensions is known to cause conflicts between adaptations —that is, whenever the value space of one dimension is contained in the value space of another dimension, this will cause conflicts during the reduction and transitive closure calculation of the system analysis.

Feature-Oriented Domain Analysis. Features models emerged with the goal of expressing distinct functionality applicable to entities on a software system.

The Feature-Oriented Programming (FOP) paradigm [54] arose to consider the implementation aspects of features. In the setting of FOP, a software system's artifacts are considered as aggregation of features, defining the main modules of the system and their functional behavior. The Feature-Oriented Domain Analysis (FODA) methodology [42] was proposed as a modeling technique to represent such modules and their sub-parts. Adaptations can be represented in feature models as possible functionality associated with a given artifact.

In the smart-home system example, the functionality associated with windows can be modeled as shown in Fig. 3. Software products are built by combining features described in the feature model starting from the root until the leaf nodes. A possible product composition of the smart-home system is a house environment consisting of the **Airco**, **Camera**, **MonitorUser**, **UnknownUser**, **Window**, **Close**, **Infrared**, **RFIDReader** features. Note, however, that not all possible combination of features would make a consistent product according to the expected behavior of the system. In the previous feature composition, if we add the **Open** feature associated with the **Window** appliance, the resulting product would be inconsistent with the requirement to lock down windows for unknown users.

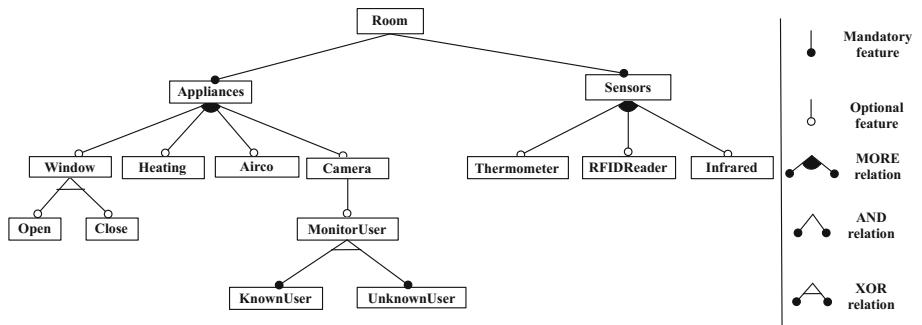


Fig. 3. Design of the smart-home system example using features.

To avoid inconsistent product configurations, feature models can be enhanced with constraints defining interaction between features. Common interaction constraints define *exclusion*, *inclusion*, or *requirement* relations between features. These constraints can be expressed by extending the feature diagram with semantic elements expressing the relations or using a textual definition [36]. Below we present two constraints defined textually. Constraint (C1) ensures that whenever the **UnknownUser** feature is selected in the product, the **Open** feature is not and viceversa. Constraint (C2) ensures that if the **KnownUser** feature is selected, then the **Open** and **Close** features are also selected.

$$\text{UnknownUser excludes Open} \quad (C1)$$

$$\text{KnownUser includes Open} \wedge \text{Close} \quad (C2)$$

Feature interaction reasoning techniques in FOP are used to identify systems that would yield inconsistencies due to the interactions of the features composing them. Traditionally, this has been treated as a static problem tackled via automated reasoning based on variety of approaches, such as CSP, model checking, computational logics, or state machines [11]. Analyzing interaction between features becomes even more challenging when composing the features at run time, in the majority of cases being an undecidable problem. Feature interaction can be identified using on-line managers [55], and negotiation agents [2]. However identification of feature interaction problems is only one piece of the puzzle. Inconsistencies caused by interactions also need to be resolved at run time. To overcome this problem, recent approaches use upfront definition of policies between features that may interact at run time [46]. Whenever a feature is included in the program at run time, the policies are verified. In case of an interaction problem, the corrective process defined by the policy associated with the conflicting features is applied.

Requirements satisfaction: In self-adaptive systems, adaptations can be abstracted as features, defining the relations between adaptations following the FODA process. As adaptations can have a direct abstraction as features, these satisfying the abstraction requirement. Inconsistency identification and resolution is based on automated reasoning approaches and predefined policy rules. As a consequence, not all possible interactions may count with a resolution strategy, causing inconsistencies to persist at run time, due to the unannounced appearance of features during execution, dissatisfying the run-time consistency requirement. Identification of inconsistencies can be performed statically, satisfying the offline consistency requirement. Nonetheless, verification and application of conflict resolution policies have been reported as costly operations, making these analysis techniques less suitable for highly dynamic settings as those proposed by self-adaptive systems.

Application to self-adaptive systems: The use of features is common in the design of Software Product Line (SPLs) since they model the concept of multiple products having a common structure. At run time, these models are still used as a basis of dynamic SPL [33], where variations of a running software are generated using incremental move or regenerative composable component construction techniques. These techniques are mainly characterized by a *synthesis* and *modification* steps. In the synthesis step, a new base model is generated using the new configuration that gathers applicable adaptations. In the modification step, the differences between the original base model and the generated model are calculated. These two steps are supported by an internal rule engine of the system, gathering adaptation policies and evaluating their feasibility from the space of possible adaptations. FODA has been used in adaptive systems in the domains of automated manufacturing [33], and critical systems [13].

3.3 Rule-Based Approaches

Rule-based approaches consist of (external) production rule systems coordinating the behavior of a software system. Rule execution engines are used to maintain a clear separation between data and business logic. The idea behind rule execution engines is to process complex rules efficiently as data is made available from internal (*e.g.*, system monitors) or external (*e.g.*, sensors) sources. Rule execution engines exhibit one of two implementations to process such data. (1) Forward-chaining engines [28] infer the validity of rules describing the system based on the available input data. Immediately after processing the data the actions described by evaluated rules are enacted. (2) Backward-chaining engines [66] infer validity of rules based on the system’s goals. A path of rules to be executed is build backwards from the goals until the rules satisfied by the input data are found. Once the path is complete, the actions described by these rules are executed.

Rule execution engines are used as coordination models providing a consistent view of software systems. The view of the system can be generated, for example, by means of a fact space [50] (using computational logic for its definition and processing).¹ Data gathered about the system’s state is modeled as *facts*, and *transition rules* describe conditions on such states. Each rule is associated with a set of actions to be taken whenever it becomes valid. At run time, the visible actions of the system correspond to those actions for which their associated rules are valid, while actions associated to invalid or inconsistent rules are oblivious to the system execution. Snippet 3 shows the description of two facts (Lines 2 and 3) and two rules (at Line 5 and Line 11) for the smart-home system example using Crime [50], a language for developing context-aware applications using a fact space model. The `:monitorUser` rule is used to monitor the status of a user if this is known. This rule is applicable if and only if there is a fact in the current fact space, registering a user with his id for which the status is known. Note that fact spaces allow us to restrict the domain in which production rules are applicable. For example, the `:changeWindowState` production rule is only applicable when the user fact is available for a specific `room` fact space (the rule matches with the window fact as given by its parameters). Rules applicable for the complete system are conditioned in the `public` fact space.

Rule-based approaches provide support for the development of self-adaptive systems by introducing adaptation entities as facts and the conditions evaluating such facts as rules. Adaptations are managed at run time by means of fact matching as specified by the rule engine. Whenever a fact in the fact space matches a rule, the adaptation is considered available and the actions described by the rule are composed into the system —that is, they are executed. Unmatched facts represent unavailable adaptations and the actions described by their rules are withdrawn from the fact space —that is, they are not executed.

¹ Graph transformation models, like AGG, can also be used as engines for rule-based approaches.

```

1 //facts
2 room -> window(livingRoom, open).
3 public -> user("Nicolas", known).
4 //rules
5 :changeWindowState(?room, ?weather, ?state) :-
6   room -> user(?id, ?status),
7   userLocation(?room),
8   ?state == windowState(userPreference(?id)),
9   ?weather == sunny
10  ?status == known
11 :monitorUser(?id) :-
12   public -> user(?id, ?status),
13   ?status == known

```

Snippet 3. Facts and production rules for the smart-home environment case-study.

Requirements satisfaction: Rule-based approaches can represent adaptations as facts, however, there is no explicit representation of interactions between adaptations. Interactions are represented implicitly as a consequence of rule matching. Since there is no explicit representation of adaptations or their interaction, we conclude rule-based approaches do not satisfy the abstraction requirement. In contrast, the natural rule matching algorithm lends itself for effective run-time management of adaptations, ensuring that new actions are available in the system if their associated rules are matched by consistent facts (adaptations), if conflict resolution algorithms are set in place, partially satisfying the run-time consistency requirement. Even if rule-based approaches can manage the run-time consistency of the system by ensuring that production rule actions only take place whenever their conditions are satisfied, it is still up to the programmer to ensure that the provided facts are correct and there is no accidental interactions, starvation, or cyclic delegations between production rules. Rule-based engines do not provide support to ensure this and therefore do not satisfy the offline consistency requirement.

Application to self-adaptive systems: Rule-based approaches are used to process/solve complex systems based on available data. Similar to CSP approaches, in order to enable self-adaptation, rule-based approaches must be extended to account for adaptations. Once the notion of adaptation has been defined and software systems can be modularly composed in different ways according to their surrounding environment (*i.e.*, available data), they can be used to ensure the system is composed consistently. For example, forward chaining engines can be extended as it was done in Crime to generate context-aware systems [50], where actions executed in rules are made available based on the validity of facts. Such an approach can be used to implement pervasive applications. Backward chaining engines can be used to enable behavior adaptation at the service composition level for distributed system, as explored in the Semantic Service Overlay Network (SSON) [22]. In SSON, services' tasks (components of a service) can be adapted at run time if task providers disappear from the network. Adaptation of a task takes place by replacing the missing task by a set of tasks providing behavior equivalent to the missing task.

3.4 Transition System Approaches

Transition systems are used to represent a system's states and the actions between them. Using such representation, software systems are said to transition from one state to another via predefined actions. Transition systems are normally represented as graph-like structures, where nodes denote the states of the system and labeled edges denote actions to transition between states. Transition systems are used as a modeling technique for software systems providing a structural representation and a formal basis to reason about their properties.

The transition system approaches described here: *automata* and *labeled transition systems*, and *statecharts*, *dataflow graphs*, and *Petri nets* are selected as these have been used in the implementation of concrete models for the assurance of consistency of self-adaptive systems.

Automata and Labeled Transition Systems. Automata [35] are graph-based models used to describe systems' behavior based on their possible states and the set of actions to be taken at each state. Automata are normally used to verify system properties, such as program termination. Automata are preferred as modeling techniques for software systems because they present a model that is easy to operate, for example, to merge, intersect, and (parallel) compose software systems. Similar to automata, Labeled Transition Systems (LTS) describe systems as sets of states and transition functions between such states. Since LTS are normally represented by means of automata we consider them jointly in the following.

Different automaton models have been used to represent and manage self-adaptive systems.

1. Using states to represent the execution flow of the system in which each state can execute a certain set of actions, and the transitions between states follow the control flow of application under development [7,37].
2. Extending states with behavioral constraints, as in $S[B]$ -Systems [65], where each possible state is expressed and the transitions between states are explicitly specified.

Two characteristics are common to these models. First, adaptations are not explicitly represented in the model, rather the different states of the system are defined, and adaptations are implicit from the actions the system can take in each state or the transition between states. Second, the “current state” is given by a single node in the graph —that is, only one of the states of an automaton is “active” at a time. This leads to explicitly representing possible state combinations in the system, leading to state explosion as the system grows.

Nonetheless, automata could be used directly to represent system adaptations, as shown for the smart-home system in Fig. 4, state transitions represent the change detected in the surrounding environment. Note that since adaptations occur unannounced and unordered, therefore, all states should be reachable from any other state, making the automaton cluttered and difficult to manage. On the contrary, inconsistent states can be explicitly represented as non final states.

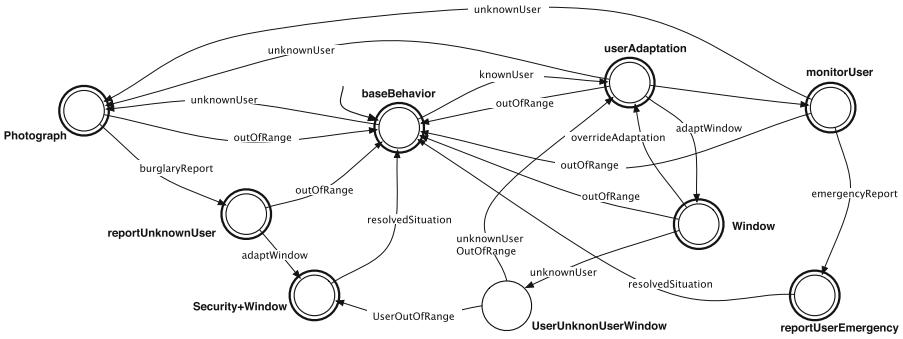


Fig. 4. Automata model for the web booking application.

A possible inconsistency in the smart-home system example can exist whenever a user's preferences are set to open the windows, and there is an unknown user, requiring the windows to be closed. In such a situation the state `UserUnknownUserWindow` would be reached in Fig. 4. On the one hand, this approach presents as a drawback that all inconsistencies must also be known beforehand by developers, which is unfeasible in self-adaptive systems. Moreover it significantly increases the size of the automaton. On the other hand, if no inconsistencies are explicitly modeled, then verifications of the system as termination become trivial (the system always terminates as all its states are final).

Requirements satisfaction: Using automata or LTS, adaptations are modeled as states of the system in which behavior specific to that state and only that state is available. Such model allows the explicit definition of adaptations, satisfying the abstraction requirement. Additionally, automata models have the advantage of providing a firsthand view of the system. At run time, this can be used to validate that the system remains in a consistent state. Furthermore, recent approaches complement automata models for self-adaptation to verify that new adaptations are consistent with respect to the initial specification of the system [37]. Bringing together these two characteristics, this approach satisfies the run-time consistency requirement. Finally, automata are commonly used due to their decision characteristics to compose automata and reason about their properties (e.g., program termination), hence, satisfying the offline consistency requirement.

Application to self-adaptive systems: As previously mentioned, recent approaches, as ActiveFORMS [37], foster the use of automata to build self-adaptive systems, applied to CPS. In this approach the system uses automata as the run-time model of the system in charge of analyzing and executing adaptations according to the system's surrounding environment. The system further presents a goal management model that allows the modification of the run-time model itself—that is, the (re)definition/deletion of adaptations and their interactions at run time. The goals of the system are specified as boolean rules and

are verified dynamically whenever the automata run-time model needs to be modified. Such verification process takes place to ensure that modification of the system's goals remain consistent with respect to its formal specification. Other approaches have used automata to model adaptive ecology systems [65], and camera detection systems for UAV [7].

Statecharts. Statecharts provide a specification of the internal behavior of system components as well as the interactions between components in the setting of reactive and event-based systems [31]. Statecharts are an extension of state transition diagrams (*i.e.*, LTS), proposed to tackle the state explosion problem by successfully modularizing components in a hierarchical way. Additionally, the model enables system designers to easily express general properties such as *clustering*, *orthogonality*, or *refinement*.

Figure 5 shows a statecharts diagram of the smart-home system example, where states are represented as rounded rectangles, events are represented as labeled arcs between states, and conditionals are represented as circles. In Fig. 5, clustering and hierarchy between states is depicted by the containment relation, inner states have a higher hierarchy than outer states. Orthogonality is depicted between the `Detected user` and `Emergency` states by means of the dotted line separating the two states. Whenever the container state is accessed, the two sub-states work independently of each other, by applying the `User adaptation` state in the first case, and the `Monitoring` state in the second case. Refinement is the property of statecharts to capture the different components of the system. For example, Fig. 5 represents the user detection component (the refinement) of a smart-home system.

The visual representation provided by statecharts accounts for the formal specification of the system. Such specification can be used to analyze different properties about the system [44].

Requirements satisfaction: Since statecharts are a structural extension of automata, they also satisfy the abstraction and offline consistency requirements. Abstraction is satisfied by defining the states of the system as adaptations in which specific behavior is available. Offline consistency is satisfied by the mechanisms to reason about system properties beforehand. This mechanism can be supported by means of model checking techniques, taking advantage of the explicit representation of the system behavior and state changes as reactions to event changes. The run-time consistency requirement is however not satisfied. In order to satisfy this requirement, statecharts would have to be extended with support to introduce new states, for example, via event triggering. Introduction of new states also requires a run-time verification process.

Dataflow Graphs. Dataflow programming [40] represents programs as directed graphs where data flows between nodes along the graph's arcs. Graph's nodes represent program entities (*e.g.*, modules, objects, or instructions, according to the level of abstraction used), and directed arcs represent data

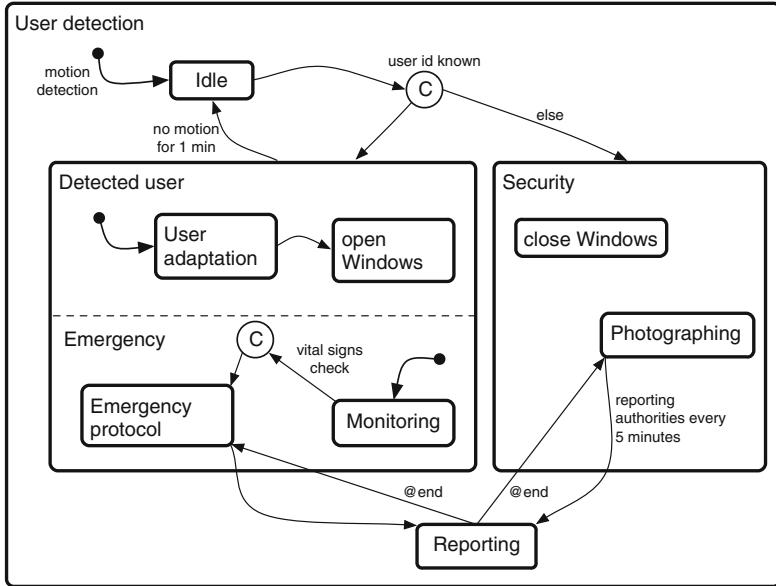


Fig. 5. Statecharts model for the user detection service.

dependencies between program entities. Dataflow graphs are used to specify the computation (*i.e.*, *are* the system) and manage the system.

Figure 6 shows the user detection example of the smart-home system expressed as a dataflow graph. For the example, we use the dataflow graphs as defined in AmbientTalk/R^V [47]. The management model of dataflow systems consists of *operators* processing input data (*e.g.*, node **RFIDSensor** processing the **movementDetected** event in Fig. 6) and producing output data (*e.g.*, node **HouseEmpty** producing the **empty** boolean value in Fig. 6). Dependencies between nodes are explicitly represented by edges in the graph. Data always flows from the outputs generated by a node to the input of another node. A node in a dataflow graph is said to be enabled to fire if there are values available for all of its inputs. Node firing (execution) can take place at one of two moments. Once *all* of its inputs have received a (new) value, or whenever *one* of its input values changes. These two approaches are referred to as *synchronous dataflow* and *asynchronous dataflow*, respectively. Arcs for nodes like **RFIDSensor** are called forking arcs. Whenever data reaches a forking arc it is duplicated and sent to each of its subsequent nodes. Dataflow graphs enable the parallel execution of processes. Tasks that can be computed in parallel are depicted by the H_i doted lines in Fig. 6. All operations in the same line fire in parallel (as long as they can fire). That is, nodes **UserIdentification** and **MonitorUser** can be computed in parallel if **RFIDSensor** produces output data.

Systems progress is unequivocally dictated by its dataflow graph. Hence, dataflow graphs successfully manage software systems. Modeling adaptations as

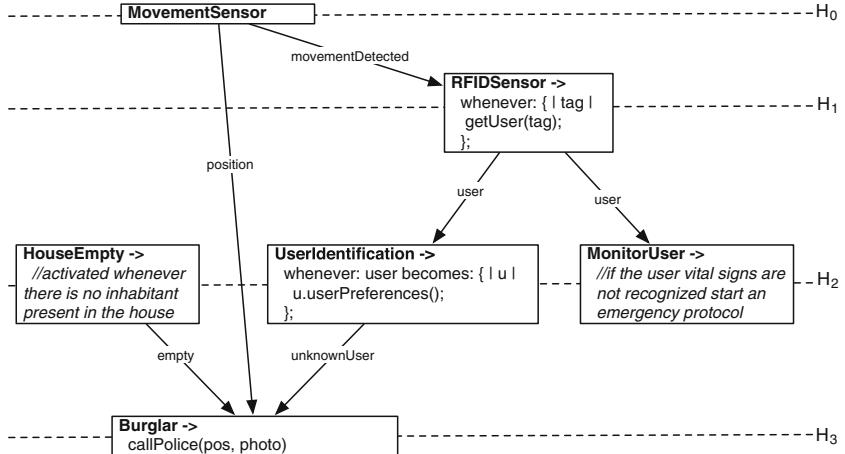


Fig. 6. Dataflow graph representation for user detection in the smart-home system.

system entities (*i.e.*, nodes in the graph), they will become available as long as the appropriate input is provided to their nodes. Adaptations are then managed using the dataflow as control. This means that if new sources of data appear in the surrounding environment, new adaptations could be added dynamically. However, dataflow graphs do not provide a direct way to analyze properties of the system. The system and its adaptations will be consistent if and only if the definition of the graph is. To verify the consistency of the graph, dataflow graphs must be extended using rule-based systems or model-checkers.

Requirements satisfaction: Dataflow graphs satisfy the abstraction requirement by defining adaptations as nodes in the graph. This provides a level of abstraction similar to that of other transition systems in this section. Dataflow graphs have been successfully used as visual programming languages, where the dataflow graph is not only a representation of the system, but it is indeed the execution model of the system to which new nodes can be added at run time. This provides an adaptive control structure for the system's data flow. Such visual run-time model of the system can be used, as a debugger [32], in dataflow programming languages like NL. However, it is not possible to verify that existing or added adaptations are consistent with each other at run time or beforehand. As a consequence, dataflow graphs do not satisfy the run-time or offline consistency requirements.

Application to self-adaptive systems: The computation style offered by dataflow graphs is of particular use in situations requiring data processing, as it is the case of reactive programming. The data processing control structure used in reactive programming is extended with adaptation specific constructs in the Flute [4] language, enabling users to define self-adaptive systems that respond to continuous input from their surrounding environment; used for example, in adaptive

personalized assistants. In Flute, developers do not directly interact with the dataflow graph but rather are exposed to language abstractions that create, manage, and modify the graph behind the scenes. However, such languages, do not directly provide verification mechanisms for consistency. In order to enable dynamic verifications the language would need to be extended to account for consistency.

Petri Nets. Petri nets [51] are directed graph structures consisting of places, transitions and tokens. Places represent the different states of the system, transitions represent actions between states and tokens represent data flowing in the net. Tokens reside in places, indicating that the place is currently active. At a given moment in time, the set of tokens in all places is defined as a marking. Tokens (data) flow from one place to another if the transition between them fires. In the basic Petri net model, transitions are enabled to fire whenever all of their input places are marked.

In Petri nets, similar to automata, adaptations can be defined as states in the net associated with specific behavior which becomes visible only when the state is active. Moreover, interaction between adaptations can be modeled by defining transitions between the states representing adaptations, so that the active state of an adaptation can influence that of other adaptations defined in the system. Figure 7 shows a Petri net model for the smart-home system example expressing the interaction between the `KNOWNUSER` (K), `WINDOW` (W), `SECURITY` (S), and `UNKNOWNUSER` (U) adaptations, following the specification of self-adaptive systems, as defined by the context Petri nets (CoPNs) [18] runtime model.

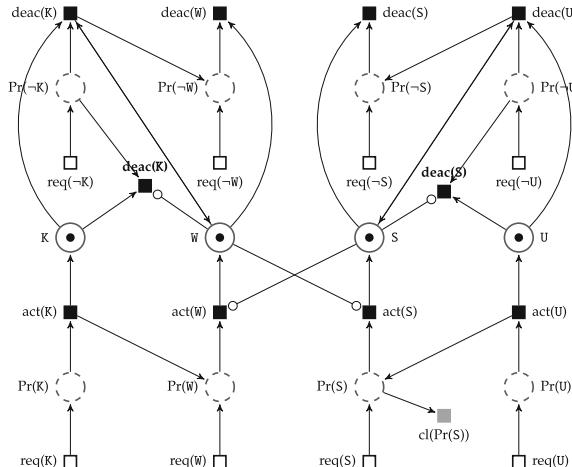


Fig. 7. Smart-home system adaptations' interaction design using CoPN.

Petri nets provide a first hand view of the model at run-time which can be used to manage the consistency of the system as adaptations become active and inactive. That is, the conditions in which an adaptation can become active or inactive can be constrained by the state of other adaptations. Such constraints are encoded as combinations of transition firings. This means constraints will be automatically verified and enforced at run time as part of the execution semantics of the Petri net (*i.e.*, flow of tokens). Additionally, Petri nets are formal specification of software systems, as such, they also enable the analysis of different system properties like liveness, deadlocks, or reachability. Using the analysis techniques already provided in Petri nets it is possible to reason about the consistency of self-adaptive systems beforehand. For example, inconsistencies in the interaction between adaptations could be identified whenever: there are deadlocks, it is no longer possible to activate a given adaptation, or it is possible to activate adaptations that should not be active in presence of other adaptations [16].

Requirements satisfaction: Petri nets combine the modeling power of automata and dataflow graphs, allowing developers to define adaptations as particular states (places) of the system, satisfying the abstraction requirement. Note that in Petri nets there is no state explosion problem, as its inherent support for concurrency allows to model different states being active simultaneously by means of the Petri net marking. Interactions between adaptations is defined through the firing of actions (transitions), which follow the operational semantics of Petri nets. Additionally, new adaptations could be introduced to the system by means of Petri net composition. Since consistency of adaptations is verified at run time as transitions fire, the model satisfies the run-time consistency requirement. Finally, the analyses provided in the Petri nets formalism allow us to reason about different system properties, satisfying the offline consistency requirement.

Application to self-adaptive systems: Petri nets have been used to model the dynamics of software systems. Their applicability to self-adaptive systems is realized, for example, in the CoPNs model [18]. This model defines a Petri net structure as the run-time model of self-adaptive systems, and it uses a programming language layer for the interaction with the model —that is, developers do not directly manipulate the Petri net, but rather interact with it through a language layer. In particular, CoPNs are designed to deal with interactions between adaptations, allowing their definition as part of the run-time model of the system. Furthermore, CoPNs define management and analysis modules for the system, in which it is possible to verify the consistent activation of adaptations at run time [17], and analyze the coherence and consistency of the system beforehand [16].

4 Analysis and Challenges of Consistency Management

Section 3 outlines different models and techniques used in the development of self-adaptive systems to assure the run-time consistency of adaptations as they

interact with each other. This section analyzes the approaches with respect to the requirements **M.1** through **M.3** put forward in Sect. 2.2 and discusses avenues for future work in the development of concrete models to manage self-adaptive system's consistency.

4.1 Analysis of Consistency Assurance Models

Whenever systems' behavior changes over time there are no guarantees that the new behavior complies with the expected behavior of the system, possibly leading to system errors or inconsistencies. It is therefore of the utmost importance to manage the adaptation process in a way that enables the system to ensure new behavior interacts correctly with that already deployed in the system. The description of the approaches presented in Sect. 3 is motivated by this observation, focusing on a development perspective for the verification and analysis of the system consistency with respect to its changing environment. Table 1 presents a summary of the described approaches and their satisfiability of the requirements for consistent interaction between adaptations (shown as black boxes in the table). In the following we describe approaches' properties that lead to the satisfaction or dissatisfaction of each of the requirements.

Table 1. Consistency interaction properties satisfaction.

Models for Consistent Adaptation Interaction		Abstraction (M.1)	Run-time (M.2)	Offline (M.3)
Formal Approaches	Computational Logic			
	Algebraic Specification		■	
	Model Checking		■	
	Constraint Solvers		■	■
Architectural Modeling Approaches	Model Transformations	■		
	Feature-Oriented Domain Analysis	■	■	
Rule-Based Approaches		■	■	■
Transition System Approaches	Automata & LTS	■		
	Statecharts		■	
	Dataflow Graphs	■		
	Petri Nets	■	■	■

M.1 Abstraction: Managing self-adaptive systems' adaptations at run time requires to model adaptations explicitly, such that adaptations' state and their interactions are well defined in the system. Such an explicit representation of adaptations, provides a structured model to dynamically manage changes in the system's states in response to its surrounding environment. Additionally, it is possible to model adaptations' interactions such that their state changes as a consequence of changes to other adaptations too. Formal

and rule-based approaches do not satisfy this abstraction requirement while architectural modeling and transition system approaches do.

Formal approaches are normally used for the specification of software systems by means of formulae. Using such specification it is then possible to reason about different system properties (beforehand). In the context of self-adaptive systems, formal approaches have been used as external models to prove properties about the system, rather than to model its behavior, or drive its development or execution. Even though formal approaches (*e.g.*, modal logics [68]) have been extended to take into account adaptation concepts, they still play an auxiliary role in the system development —that is, are used for specification and analysis purposes. As a consequence, systems' consistency is limited by its specification. Particularly, expressing adaptations interaction as formulae, requires to model *every* possible interaction by a formula, which can become cumbersome or possibly unfeasible for large-scale systems if no automated support to generate such formulae is provided, the assurances about the system's consistency are limited. These reasons lead us to conclude that, however useful for supporting self-adaptive systems development, formal approaches do not provide the necessary abstractions to develop consistent self-adaptive systems.

Architectural modeling approaches provide a structured view of the system at design time. In this, view the main entities of the system have an explicit representation in a graphical notation, that can be mapped into the system's development environment. Similar to other system entities, adaptations can be defined as independent model entities, while interactions between adaptations are explicitly defined by means of rules expressing how other system entities (and adaptations) are affected by a new adaptation. Such explicit representation of adaptations in a system model can be used for the development of the system, however, a transformation between the model and its implementation (*e.g.*, using ATL or FOP) would be required. Additionally, not all rule systems defining interactions are suitable for these approaches. Rule systems used are required to have a transitive closure property, so that not all interactions need to be defined explicitly, for example, as is the case of FODA approaches.

Rule-based approaches (partially) abstract the state of the system by the use of facts over which it is possible to reason about by means of rules. Facts are statements about the state of variables in the system, while rules are used to take dedicated actions upon fact's validity. Rule systems do not provide the means to define adaptations explicitly, rather adaptations need to be regarded as particular facts that are only valid under certain situations. Additionally, definition of interactions between adaptations is not explicitly supported in rule-based systems, rather evaluation of rules generates implicit dependencies between them.

Transition system approaches effectively abstract the states and actions of the system by representing them as graph-like models, where graph nodes represent adaptations and interactions between such adaptations are represented

as edges between nodes. In these approaches, the state of adaptations depends on whether the graph node representing the adaptation is the current node of execution. The advantage of using transition system approaches is that they provide a first-hand view of the system. Moreover, these approaches can be mapped directly to the execution environment of the system, thus, there is no need for additional transformations of the transition system specification, as is the case in model-based architectural approaches.

We recognize that transition system approaches could better model self-adaptive systems. In transition system approaches, adaptations and their interactions can be naturally abstracted as states of the system and actions between those states, respectively. Such abstractions provide a first-hand view of the system that can be effectively used during its execution to orchestrate operations (*e.g.*, composition, inclusion, removal) over adaptations, as well as its verification.

M.2 Run-time Consistency: The run-time consistency requirement refers to the ability of software systems to assure that newly introduced adaptations do not break the original system behavior, or that already provided by other adaptations. To satisfy this requirement, an adaptation management approach must allow the run-time introduction of adaptations. Moreover, if the approach provides a means to reason about system properties, such mechanism is to be triggered whenever adaptations are introduced to or withdrawn from the run-time environment. Ensuring the run-time consistency of adaptations is principally supported by formal and rule-based approaches. In the former case using dynamic verification of the system's specification, and in the latter case using the execution of rules defining the control flow of the system. Note, that two of the transition system approaches, namely automata and Petri nets, also satisfy this requirement. This is due to the execution semantics accompanying these models, providing a dynamic behavior similar to that specified by rule-transition systems.

Formal approaches express system specifications in terms of formulae. Such formulae are evaluated using the techniques developed in each approach in order to prove satisfiability of the system's properties. Formulae may specify features, interactions, or constraints in a system. The general technique used in formal approaches (except for algebraic specifications which do not satisfy the run-time consistency requirement) to guarantee the system's consistency as adaptations are introduced to and withdrawn from the system (*i.e.*, by adding or removing formulae) is to verify the satisfiability of the formulae set whenever this changes. If the set of formulae is large, the verification process may be time-consuming, therefore, formulae verification is normally performed statically. In recent years, researchers have explored techniques to speed up the verification process in order to use verification techniques at run time. Some of the formalisms in which run-time techniques can be used include computational logic and model checking [68], and CSP [59].

Architectural modeling approaches are defined statically, where all adaptations are defined during the system's design. The run-time management featured in

these approaches extends to the composition of defined adaptations. If new adaptations are introduced at run time, these are not verified. Any inconsistency that may have been introduced due to interactions with new adaptations, or may have been overlooked during the system's design, will perdure at run time. For example, whenever the domains of adaptations' attribute values have a containment relation, errors might appear at run time [23].

Rule-based approaches can publish facts and production rules at any moment in time. Once a production rule is published into the fact space, it is automatically evaluated with all existing production rules. Changes in the behavior of the system (*i.e.*, adaptations) are affected immediately by the validity of the new facts associated to it. However, the consistency of the system relies on programmers to ensure facts are correct and there are no accidental interactions between facts.

Transition system approaches, similar to architectural modeling approaches, are used to model states and actions of a system. As such, these approaches can only provide verification of system properties if the approach's underlying specification allows it to do so, as it is the case of automata and Petri nets. Automata composition (*i.e.*, union, product, intersection) may preserve the properties of its components. For example, union of regular automata is a regular automata. In the Petri nets case, properties such as liveness or deadlocks can be verified at run time, for example, as new places and transitions are added. Similar verification mechanisms could be defined for statecharts or dataflow graphs, although these are currently not implemented in the concrete models surveyed. Unlike architectural approaches, Automata and Petri nets are defined with an execution semantics that enable the verification of the semantics of the system at run time. That is, if adaptation's interactions are encoded within in the model, it can be assured that the desired interactions are maintained (*e.g.*, CoPN [16]).

Formal approaches are the ideal candidate to enable the run-time verification of self-adaptive systems. Formal approaches encode properties about a system (*e.g.*, consistency) as part of their specification, allowing the system to re-evaluate such specification whenever it changes.

M.3 Offline Consistency: The offline consistency requirement refers to the ability to reason about the system and its properties at early stages of the development cycle. Upfront analysis of systems' properties is motivated by two observations from the conclusions of the run-time consistency requirement. First, run-time verification of all system properties may be too time consuming for medium- or large-scale systems. Second, the run-time verification process relies on the composition properties of the model. For example, it should be the case that, given two consistent adaptations their composition will be consistent. The support that upfront analysis can provide is to reduce the time of verification at run time and to assure initial adaptations are consistent. Most of the surveyed approaches provide a means to reason about systems' properties beforehand, in some cases by complementing the approach with other of the approaches proposed.

Formal approaches are by definition reasoning frameworks. The ability to analyze system properties is inherent to these approaches, be it by formal verification or theorem proving. Properties, like consistency, must be encoded into the formalism taking into account the relations describing interaction between adaptations. Once these have been defined, satisfiability of such properties can be analyzed. It is however unclear, in some cases, how to capture the run-time properties of the system by means of a formal model, in particular if the model's main focus is not on the run-time behavior of the system. Such a problem can be solved by complementing the formal approach with an approach providing a run-time representation of the system, as transition system approaches do. Common properties verified by formal approaches include, system liveness and deadlocks, program invariants, and progress and safety of future states.

Architectural modeling approaches are introduced with the sole purpose of providing the means to reason about system properties using dedicated analyses. These analyses use the abstract model of the system to identify inconsistencies between defined system entities (*e.g.*, adaptations) and their possible composition or interaction at run time. Analyses in architectural modeling approaches are normally supported in combination of formal approaches, for example, model checking or constraint solving techniques to analyze the system beforehand.

Rule-based approaches are not concerned with the analysis of system properties beforehand, so no offline consistency support is provided for any kind of system property. These approaches are intended as reactive approaches in which system rules and facts are evaluated as they are needed, or become available. If inconsistencies would arise from rules' definition or evaluation, it is up to the programmer to identify and solve them.

State machine approaches provide support for structural system properties, such as composition or hierarchization. Even though they provide only limited support for the analysis of run-time properties of the system, the supported analyses could be used beforehand to reason about properties such as reachability, timeliness, liveness, or coherence between adaptations. As it is the case with architectural modeling approaches, the analysis of transition system approaches is normally supported by formal approaches, like model checking or CSP.

The majority of the approaches presented here provide means to analyze the system properties beforehand, thus many of them would be suitable to assure the consistency of self-adaptive systems. However, as noted, some of the approaches rely on analysis techniques provided by formal approaches. As a consequence, formal approaches can be put forward to enable consistency analysis of self-adaptive systems.

Three things are apparent from Table 1. A first observation is that all approaches satisfy either the run-time or offline consistency requirements. This means that, up to some extend, existing approaches for self-adaptation are concerned with managing interactions consistently. A second observation is that

formal approaches are naturally conditioned for verification and analysis (respectively satisfying the run-time and offline consistency requirements) of software systems. Nonetheless, the importance of an explicit representation of adaptations (the abstraction requirement) triumphs as a vital characteristic for the development of self-adaptive systems. Finally, with the exception of automata and Petri nets, none of the approaches satisfy all three requirements. As a consequence, in order to satisfy all three requirements it is likely that two or more of the described approaches need to be combined in a concrete model, as it is the case for some of the concrete models discussed in Sect. 3.

In particular, we note there is a close relation between transition system approaches and formal approaches, where the former are used for the explicit representation and run-time management of adaptations, and the latter provides the underlying specification of the system. Such specification is used to assure consistency properties about the system through dedicated analyses at development time, and run-time verification during execution. There is also a strong connection between architectural approaches and rule-based approaches. Where the former provide a concrete representation of the system and its adaptations, as well as analysis mechanism to reason about system properties (*e.g.*, consistency between model artifacts). Note that analyses in architectural approaches are usually supported by formal approaches like logic programming [19] or description logics [67]. Rule-based approaches can be used to manage and actuate adaptations at run time, as for example is the case of AGG in [23]. Finally, approaches like discrete control theory [45] also live at the intersection of the different approaches discussed here. Discrete control systems can be used to deal with inconsistencies [3] in autonomous systems or to assure the correctness of adaptive component systems [1]. Discrete control systems are realized by a set of automata structures specifying the main system and its properties. Graph transition systems are used to combine the different automata and enabling/disabling specific behavior in the system. Consistency or correctness properties of the system are verified using a logic system describing the system and its general goal, much in the way model-checking approaches work.

It is important to recognize that although satisfiability of the consistency assurance requirements can be due from the combination of transition system approaches with formal approaches, such combination is not a necessary condition. Concrete models, such as ActivFORMS [37], context Petri net [17], and reactive control techniques [5], integrate verification and analysis mechanisms into the development environment in order to manage interaction between adaptations at run time, and to reason about system properties beforehand, respectively.

Table 2 presents a summary highlighting the trade-offs of using the different approaches presented in this chapter for assuring consistency of adaptations interactions as part of the development process of self-adaptive systems.

Table 2. Summary of consistency assurance approaches for self-adaptive software systems.

Approach	Characteristics
Computational Logic	<ul style="list-style-type: none"> ✓ AST representation of the system control flow ✓ Logic specification used for the analysis of system properties. e.g., temporal properties with LTL ✗ Run-time verification requires the user to be in the loop ✗ No explicit representation of adaptations
Algebraic Specification	<ul style="list-style-type: none"> ✓ Properties verification limited to that of the specification's proof ✗ No explicit representation of system entities
SMT - Model Checking	<ul style="list-style-type: none"> ✓ System properties specification through logic formulae ✓ Supported by a state transition model to solve the system ✓ Verification can be postpone until run time, reusing results gathered offline ✗ No explicit representation of adaptations
SMT - Constraint Solvers	<ul style="list-style-type: none"> ✓ Constraint verification as the system environment changes ✗ No explicit representation of adaptations
Model Transformations	<ul style="list-style-type: none"> ✓ Objects and adaptations are represented as high-level model entities ✓ System properties are verified offline supported by model verification techniques e.g., model checking ✗ Interactions between adaptations need to be made explicit beforehand ✗ No consistency verification at run time
Feature-Oriented Domain Analysis	<ul style="list-style-type: none"> ✓ Expressing adaptations and their relations as a feature model ✓ Interactions between adaptations are verified offline, and can be rechecked at run time (only for pre-defined adaptations) ✗ No verification of incoming adaptations after the system has been defined
Rule-Based	<ul style="list-style-type: none"> ✓ Manage the dynamics of the system assuring adaptations are preserved ✗ Adaptations are not expressed explicitly ✗ All interactions and adaptations need to be expressed beforehand
Automata & LTS	<ul style="list-style-type: none"> ✓ First-hand view of the system and its adaptations ✓ Run-time management of adaptation interactions ✓ Integrated analyses to assure system properties offline; e.g., livelocks or deadlocks ✗ All system states and their interactions may need to be defined beforehand
Statecharts	<ul style="list-style-type: none"> ✓ Modular and compact representation of adaptations ✓ Offline verification of the system supported by model checking techniques ✗ No support for run-time verification of system properties
Dataflow Graphs	<ul style="list-style-type: none"> ✓ Representation of the adaptations as graph nodes ✗ The dynamics of the system are managed by the approach reactively, not verifying any properties at run time.
Petri Nets	<ul style="list-style-type: none"> ✓ First-hand view of the system and its adaptations ✓ Run-time management of adaptation interactions ✓ Integrated analyses to assure system properties offline; e.g., livelocks or deadlocks

4.2 Challenges in Self-adaptation Consistency

Current efforts in consistency assurance for self-adaptive systems foster the management of adaptations as they are introduced to and withdrawn from a running system. The concrete consistency assurance models presented in Sect. 3 are a starting point to ensure their run-time consistency. However, as it is possible to see, these models assume a central entity managing the consistency for the entire system, or take into account no time restrictions for the adaptation process. Such assumptions restrict the applicability of the models, since the target domain of self-adaptive systems is large-scale, heterogeneous and distributed environments, as for example, time-constrained and critical systems.

In order to fully embrace the vision of self-adaptive systems, we recognize that there are challenges that still need to be addressed to assure the consistency of self-adaptive systems in large-scale, heterogenous and distributed environments. We discuss these challenges and possible directions about how to address them.

Efficiency. The adaptation process proposed in self-adaptive systems is based on the MAPE loop (*i.e.*, Monitoring, Analysis, Planing, and Execution). This entails that an active monitor entity observes changes in the system’s surrounding environment; once changes are sensed, the analysis and planning phases follow to assess adaptations applicability; finally an execution phase enacts the system adaptations. The approaches described in this chapter are used during the analysis and planning phases to manage the consistent interaction between adaptations. However, as systems grow, the adaptation process can become too time consuming, lagging the execution of the system as a result.

The concrete implementations of the different approaches studied in this chapter present small and medium application examples in which execution lagging can be disparate as it has no visible impact on the usability of the system. However, it is unclear whether an impact can be observed for large systems in which the analysis and planning phases may take several seconds or even minutes, therefore hampering system’s usability. Responsiveness is crucial in critical or time-constrained systems, in which delaying execution may have repercussions on the execution of the whole system. This observation follows the requirements defined for the consistency assurance for the development of self-adaptive systems put forward in Sect. 2.2. To prove responsive, consistency assurance models must perform an initial system analysis offline (**Requirement M.3**), and can continue verifying the system at run time (**Requirement M.2**) [12].

Incremental verification [58] techniques are introduced to reduce analysis and verification execution time following the two-step model of the run-time and offline consistency requirements. The purpose of incremental verification is to reduce, as much as possible, the time it takes to verify the system properties upon run-time changes to its surrounding environment. As a consequence, these models allow to provide a system specification with incomplete information. An initial analysis is performed with the information available offline, continuing the verification dynamically as missing information is filled in at run time. Incremental verification techniques can be implemented by reusing state machine approaches as the structured specification of the system, where the variable parts of the system are verified at run-time reusing previously obtained information from a model checker [39], or probabilities are associated with unknown information beforehand, verifying such parts of the system as information becomes available (using a probabilistic Markov chain) [6, 27]. Such techniques for incremental verification are used, respectively, to efficiently verify the structure of event- and component-based systems as they change dynamically [39], and to assure the reliability of software systems as they adapt to their surrounding environment [26].

Concurrency. During the evaluation of consistency management models for self-adaptive systems we considered the adaptations to be processed sequentially. As changes are sensed in the environment, they are processed atomically and one by one —that is, processing one adaptation will not start until the processing of the previous one has finished. This method proved successful for

the models at hand in that it is possible to manage and verify incoming adaptations at run time. However, such model of computation restricts the applicability and expressiveness of self-adaptive systems, not being able to cope with large-scale and distributed environment, where changes to the environment take place simultaneously.

In self-adaptive systems, adaptations take place in response to continuous updates from a sensor network gathering information about the system's surrounding environment (*context*) and internal state (*self*). In a simplified model, each adaptation can be thought to be associated with a single sensor. Depending on the information provided by sensors, adaptations may be composed into or withdrawn from the system. Sensor information updates take place unannounced, and thus there is no single synchronization point to process gathered information. Moreover, two or more sensors could signal information updates simultaneously. Managing simultaneous adaptation requests sequentially presents two main problems. (1) Unpredictability of the system is raised. The order in which adaptation requests are processed may differ for different runs of the system. Such ordering can have an effect on the overall decision to compose the adaptations. Different sub-sets of adaptations may be composed in the system for a given set of simultaneous requests, raising the unpredictability of the system's observed behavior. (2) Incoming adaptations verification. As adaptations are introduced composition of other interacting adaptations can be affected by other introduced adaptations with which the former adaptations may interact. The outcome of the consistency verification may change depending on the moment in which the evaluation is performed. If other interacting adaptations are being composed or withdrawn, the verification may yield different results. In addition to these problems, we note that concurrent management and evaluation of adaptations becomes ever more important in the context of distributed systems, as the likelihood of multiple sensors signaling updates simultaneously increases. Based on these observations we put forward the need to evaluate adaptation requests concurrently in the concrete models for consistency assurance.

In order to embrace concurrent evaluation and verification of adaptations at run time, the consistency assurance models have to be extended with a concurrent programming model. Although some of the surveyed approaches are already equipped with the notion of concurrency (*e.g.*, Process algebras, Dataflow graphs, and Petri nets) and could therefore be used to deal with concurrency, the concrete models currently used in self-adaptive systems do not use this property for the processing, evaluation, or verification of adaptations. The impact of extending the models to deal with concurrent evaluation of adaptations is currently unknown and needs to be assessed.

Distribution. Self-adaptive systems are not always contained in a single machine. Target application domains for self-adaptive systems are usually large-scale systems composed of multiple distributed components that interact with each other. In such systems, some of the components may be unknown to a component. This may be due to network failures or components' mobility.

The approaches surveyed here assume an adaptation model with complete knowledge of the system. However, assuming such a model to manage adaptations' consistency is incorrect in the context of distributed systems.

To deal with distribution, adaptive systems need to account for distributed features at their core (*i.e.*, discovery, communication, failures). Current self-adaptive system models [15, 62] provide such features to enable self-adaptation in a distributed system. Nonetheless, to manage the consistency between adaptations, extensions to consistency management approaches are required. In a distributed setting, where the totality of the system may be unknown, each distributed component is required to manage and assure the consistency of its own adaptations. This implies having an independent management and verification module for each component. Furthermore, the consistency management modules need to be extended to collaborate with other management modules that may be discovered at run time.

Incremental management and verification techniques are commonly used in distributed systems to reason about their properties. Similar techniques could also be used to manage the consistency between adaptations in self-adaptive systems. Similar to the incremental verification processes envisioned to tackle the efficiency challenge, each distributed component is in charge of ensuring its own consistency. As other components are discovered, the consistency managers need to synchronize (an re-evaluate) to ensure the complete system (*i.e.*, discovered components) is consistent, possibly re-using the results previously obtained for each independent component. As mentioned previously, working with distributed systems evidences even more the requirement to deal with concurrency as adaptations can be requested simultaneously. However, note that as each component processes its own adaptation requests, dealing with simultaneous requests is resolved at the inter component synchronization level. Known concurrency synchronization techniques (*e.g.*, barriers) could be applied to deal with simultaneous adaptation requests in distributed components.

Support. From the surveyed approaches, it is possible to observe that the concrete models used to assure consistency between adaptations provide the support to deal with the consistency requirements. Nonetheless, for the most part, these models are detached from other parts of self-adaptive systems construction (*e.g.*, language, tool support). Specification, analysis, verification and management of consistency in self-adaptive systems should not represent an overhead for the developers of the system, as expressed by Requirement M.1 in Sect. 2.2. Developers should only be required to work with a comprehensive set of tools allowing them to build, test, and verify their system.

We recognize that to raise acceptance of self-adaptive systems, a more involved development process is required. This encompasses both a seamless integration between all phases of self-adaptive systems (*e.g.*, specification, development, verification, analysis, testing) and better tool support for each of these phases. Existing tools, for example CoPNs, already present a step forward in this

direction. Such existing tools need to be complemented by adding other system properties to be verified.

5 Conclusion

Self-adaptive systems are introduced to enable and manage the dynamic adaptation of software systems' structure and behavior. Adaptations are applied to the system in response to information gathered from a sensor network. The goal behind dynamic adaptations is to provide a more appropriate system's behavior with respect to the situations of its surrounding execution environment and to improve the system's Quality of Service (QoS) requirements. However, dynamic modification of a system's behavior does not come without complications. As behavior changes at run time, the actual behavior composed in the system may present errors due to, amongst others, inconsistencies in the interactions between its adaptations.

This chapter provides an overview of different general approaches that can be used to assure adaptations' consistent interaction in self-adaptive systems. We classified existing approaches in four categories, formal, architectural modeling, rule-based, and transition system approaches, according to the level of abstraction in which the approach is developed. To compare the approaches with one another, we put forward three requirements for the management and assurance of consistency—that is, abstraction, run-time consistency, and offline consistency. These requirements were extracted from prototypical self-adaptive systems examples, and driven by a concrete case of smart-home systems.

The observations of our comparison revealed that most of the approaches provided support for one or two of the consistency requirements but nearly half of them fail to support the abstraction requirement. As a matter of a fact, only two of the approaches satisfy all three requirements. Notwithstanding, it is also possible to identify complementary features between the four categories, where requirements dissatisfied by one approach, could be satisfied by combining it with another. Formal approaches provide the strongest foundations to assure consistency between adaptations, as they offer analysis and verification techniques to reason about the system properties. However, such approaches are only used as additional artifacts to the development of self-adaptive systems. Architectural modeling approaches are ideal for designing the system as they provide specific abstractions for the definition of adaptations. Using the specification of the system design, it is possible to reason about its properties offline. Nonetheless, there are no means to reason about the system at run time since only those entities preconceived during the design of the system are part of the reasoning process. In contrast, rule-based approaches are designed to manage the dynamics of software systems. However, the only abstraction offered by these approaches is that of facts, which can be verified at run time for satisfiability. Moreover, in rule-based systems, it is up to the developer to assure the system is consistent, as there are no integrated offline reasoning mechanisms. Finally, transition system approaches are ideal for abstracting adaptations, providing a firsthand view of

the system's structure. Depending on the specific model used, the dynamics of the system can be specified such that consistency is verified as the system progresses. Moreover, some of these approaches, supported by formal approaches (*e.g.*, model checking) can use the system abstraction as a specification to reason about system properties offline. However, not all concrete models provide such functionalities.

As a result of these tradeoffs presented all approaches, we foresee successful consistency assurance approaches to combine transition systems and formal approaches since they offer an explicit representation of the system, and the means to verify and reason about different system properties, respectively. The combination of these two approaches can be observed in the concrete models that do support all three requirements, automata and Petri nets [16, 26, 37, 39]. In addition to the analysis of consistency assurance models, we also identified the relevance of the models with respect to their application domains.

Self-adaptive systems are envisioned to operate in large-scale distributed environments interconnecting multiple subsystems, and gathering information from multiple sources. Such inherent characteristics of self-adaptive systems revealed four challenges that need to be addressed to successfully assure consistency between adaptations with respect to their target systems. Each of the challenges, efficiency, concurrency, distribution, and support, constitute an interesting avenue to explore, in which a combination of transition systems and formal approaches (*e.g.*, respectively using concurrent processing models, and incremental verification) can further complement a concrete model for consistency assurance of self-adaptive systems.

Acknowledgements. This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Center (www.lero.ie). We thank the anonymous reviewers for their comments on earlier versions of this paper.

References

1. Alvares De Oliveira Jr., F., Rutten, É., Seinturier, L.: Behavioural model-based control for autonomic software components. In: 12th IEEE International Conference on Autonomic Computing, ICAC 2015. IEEE, Grenoble, July 2015
2. Amer, M., Karmouch, A., Gray, T., Mankovskii, S.: Policies for feature interaction resolution. In: de Souza, J.N., Boutaba, R. (eds.) Managing QoS in Multimedia Networks and Services, MMNS2000. IFIP AICT, vol. 54, pp. 207–223. Springer, Boston (2000). https://doi.org/10.1007/978-0-387-35532-0_15
3. An, X., Delaval, G., Diguet, J.-P., Gamatié, A., Gueye, S., Marchand, H., de Palma, N., Rutten, E.: Discrete control-based design of adaptive and autonomic computing systems. In: Natarajan, R., Barua, G., Patra, M.R. (eds.) ICDCIT 2015. LNCS, vol. 8956, pp. 93–113. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14977-6_6

4. Bainomugisha, E., Vallejos, J., De Roover, C., Lombide Carreton, A., De Meuter, W.: Interruptible context-dependent executions: a fresh look at programming context-aware applications. In: Proceedings of the ACM International Symposium on New Ideas and Reflections on Software, OnWard 2012, pp. 67–84. ACM, New York (2012)
5. Berthier, N., Rutten, É., De Palma, N., Gueye, S.M.K.: Designing autonomic management systems by using reactive control techniques. *IEEE Trans. Software Eng.*, 18, December 2015
6. Bianculli, D., Filieri, A., Ghezzi, C., Mandrioli, D.: Incremental syntactic-semantic reliability analysis of evolving structured workflows. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014. LNCS, vol. 8802, pp. 41–55. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45234-9_4
7. Bicocchi, N., Vassev, E., Zambonelli, F., Hinckey, M.: Reasoning on data streams: an approach to adaptation in pervasive systems. In: Vinh, P.C., Vassev, E., Hinckey, M. (eds.) ICTCC 2014. LNICST, vol. 144, pp. 23–32. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15392-6_3
8. Blalckburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press, Cambridge (2001)
9. Börger, E., Stärk, R.: Abstract State Machines. Springer, Heidelberg (2003), <https://doi.org/10.1007/978-3-642-18216-7>
10. Braileford, S.C., Potts, C.N., Smith, B.M.: Constraint satisfaction problems: algorithms and applications. *Eur. J. Oper. Res.* **119**(3), 557–581 (1999)
11. Calder, M., Kolberg, M., Magill, E.H., Reiff-Marganiec, S.: Feature interaction: a critical review and considered forecast. *Comput. Netw.* **41**, 115–141 (2003)
12. Calinescu, R., Ghezzi, C., Kwiatkowska, M., Mirandola, R.: Self-adaptive software needs quantitative verification at runtime. *Commun. ACM* **55**(9), 69–77 (2012)
13. Capilla, R., Hinckey, M., Díaz, F.J.: Collaborative context features for critical systems. In: Proceedings of the Ninth International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS 2015, pp. 43:43–43:50. ACM, New York (2015)
14. Cardozo, N.: Identification and Management of Inconsistencies in Dynamically Adaptive Software Systems. Ph.D. thesis, Université catholique de Louvain - Vrije Universiteit Brussel, Louvain-la-Neuve, Belgium, September 2013
15. Cardozo, N., Clarke, S.: Context slices: Lightweight discovery of behavioral adaptations. In: Proceedings of the Context-Oriented Programming Workshop, COP 2015, pp. 2:1–2:6. ACM, July 2015
16. Cardozo, N., González, S., Mens, K., Van Der Straeten, R., D'Hondt, T.: Modeling and analyzing self-adaptive systems with context petri nets. In: Proceedings of the Symposium on Theoretical Aspects of Software Engineering, TASE 2013, pp. 191–198. IEEE Computer Society, July 2013
17. Cardozo, N., González, S., Van Der Straeten, R., Mens, K., Vallejos, J., D'Hondt, T.: Semantics for consistent activation in context-oriented systems. *J. Inf. Softw. Technol.* **58**, 71–94 (2015)
18. Cardozo, N., Vallejos, J., González, S., Mens, K., D'Hondt, T.: Context petri nets: enabling consistent composition of context-dependent behavior. In: International Workshop on Petri Nets and Software Engineering, PNSE 2012, 25–26 June 2012. Springer (2012). Co-located with the International Conference on Application and Theory of Petri Nets and Concurrency

19. Castro, S., De Roover, C., Kellens, A., Lozano, A., Mens, K., D'Hondt, T.: Diagnosing and correcting design inconsistencies in source code with logical abduction. *Sci. Comput. Program. Special Issue Softw. Evol. Adapt. Variability* **76**(12), 1113–1129 (2010)
20. Cetina, C., Giner, P., Fons, J., Pelechano, V.: Autonomic computing through reuse of variability models at runtime: the case of smart homes. *Computer* **42**(10), 37–43 (2009)
21. Chang, X., Cheung, S.C., Chan, W.K., Ye, C.: Partial constraint checking for context consistency in pervasive computing. *ACM Trans. Software Eng. Methodol.* **19**(3), 1–61 (2010)
22. Chen, N., Clarke, S.: A dynamic service composition model for adaptive systems in mobile computing environments. In: Franch, X., Ghose, A.K., Lewis, G.A., Bhiri, S. (eds.) ICSOC 2014. LNCS, vol. 8831, pp. 93–107. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45391-9_7
23. Degrandart, S., Demeyer, S., den Bergh, J.V., Mens, T.: A transformation-based approach to context-aware modelling. *Softw. Syst. Model* **13**, 1–18 (2012)
24. Devillers, R., Klaudel, H., Koutny, M.: Context-based process algebras for mobility. In: Proceedings of the 4th International Conference on Application of Concurrency to System Design, ACSD 2004. IEEE Computer Society (2004)
25. Dimitrovici, C., Hummert, U., Petrucci, L.: Semantics, composition and net properties of algebraic high-level nets. In: Rozenberg, G. (ed.) ICATPN 1990. LNCS, vol. 524, pp. 93–117. Springer, Heidelberg (1991). <https://doi.org/10.1007/BFb0019971>
26. Filieri, A., Ghezzi, C., Tamburrelli, G.: A formal approach to adaptive software: continuous assurance of non-functional requirements. *Formal Aspects Comput.* **24**(2), 163–186 (2011)
27. Forejt, V., Kwiatkowska, M., Parker, D., Qu, H., Ujma, M.: Incremental runtime verification of probabilistic systems. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 314–319. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35632-2_30
28. Forgy, C.L.: On the efficient implementation of production systems. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1979)
29. Geihs, K., Reichle, R., Wagner, M., Khan, M.U.: Modeling of context-aware self-adaptive applications in ubiquitous and service-oriented environments. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 146–163. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_8
30. González, S., Cardozo, N., Mens, K., Cádiz, A., Libbrecht, J.-C., Goffaux, J.: Subjective-C: Bringing context to mobile platform programming. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 246–265. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19440-5_15
31. Harel, D.: Statecharts: a visual formalism for complex systems. *Sci. Comput. Program.* **8**, 231–274 (1987)
32. Harvey, N., Morris, J.: NI: a parallel programming visual language. *Aust. Comput.* **28**(1), 2–12 (1996)
33. Helleboogh, A., Weyns, D., Schmid, K., Holvoet, T., Schelfhout, K., Van Bets-brugge, W.: Adding variants on-the-fly: modeling meta-variability in dynamic software product lines. In: Proceedings of the International Workshop on Dynamic Software Product Lines, DSPL 2009, pp. 18–27. Carnegie Mellon University, August 2009
34. Hennessy, M.: Algebraic Theory of Processes. The MIT Press, Cambridge (1988)

35. Hopcroft, J.E., Ullman, J.: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading (1979)
36. Hubaux, A., Boucher, Q., Hartmann, H., Michel, R., Heymans, P.: Evaluating a textual feature modelling language: four industrial case studies. In: Malloy, B., Staab, S., van den Brand, M. (eds.) *SLE 2010. LNCS*, vol. 6563, pp. 337–356. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19440-5_23
37. Iftikhar, M.U., Weyns, D.: Activforms: active formal models for self-adaptation. In: Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems, *SEAMS 2014*, pp. 125–134. ACM, New York (2014)
38. Jacobs, B.: Exercises in coalgebraic specification. In: Backhouse, R., Crole, R., Gibbons, J. (eds.) *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. LNCS*, vol. 2297, pp. 237–281. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-47797-7_7
39. Johnson, K., Calinescu, R., Kikuchi, S.: An incremental verification framework for component-based software systems. In: Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering, *CBSE 2013*, pp. 33–42. ACM, New York (2013)
40. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. *ACM Comput. Surv.* **36**(1), 1–34 (2004)
41. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. *Sci. Comput. Program.* **72**(12), 31–39 (2008). <http://www.sciencedirect.com/science/article/pii/S0167642308000439>. Special Issue on Second Issue of Experimental Software and Toolkits (EST)
42. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute, November 1990
43. Laddaga, R.: Self adaptive software problems and projects. In: Proceedings of the Second International IEEE Workshop on Software Evolvability, *SOFTWARE-EVOLVABILITY 2006*, pp. 3–10. IEEE Computer Society, Washington, DC (2006)
44. Latella, D., Majzik, I., Massink, M.: Towards a formal operational semantics of UML statechart diagrams. In: International Conference on Formal Methods for Open Object-Based Distributed Systems, *FMOODS 1999*, pp. 465–482. Kluwer, B.V., Deventer (1999)
45. Lee, T., Leok, M., McClamroch, N.H.: Discrete control systems. In: *Mathematics Complexity and Dynamical Systems*, pp. 143–159 (2011)
46. Liu, Y., Meier, R.: Resource-aware contracts for addressing feature interaction in dynamic adaptive systems. In: International Conference on Autonomic and Autonomous Systems, pp. 346–350 (2009)
47. Lombide Carreton, A.: Ambient-Oriented Dataflow Programming for Mobile RFID-Enabled Applications. Ph.D. thesis, Vrije Universiteit Brussel, Pleinlaan 2, Brussels, Belgium, October 2011
48. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes (parts i and ii). *Inf. Comput.* **100**(1), 1–77 (1992)
49. Morin, B., Barais, O., Jezequel, J.M., Fleurey, F., Solberg, A.: Models@ run.time to support dynamic adaptation. *IEEE Comput.* **42**(10), 44–51 (2009). <https://doi.org/10.1109/MC.2009.327>
50. Mostinckx, S., Scholliers, C., Philips, E., Herzeel, C., De Meuter, W.: Fact spaces: coordination in the face of disconnection. In: Murphy, A.L., Vitek, J. (eds.) *COORDINATION 2007. LNCS*, vol. 4467, pp. 268–285. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72794-1_15

51. Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989)
52. Pinna Puissant, J., Van Der Straeten, R., Mens, T.: Badger: a regression planner to resolve design model inconsistencies. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 146–161. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31491-9_13
53. Prasad, M.R., Biere, A., Gupta, A.: A survey of recent advances in sat based formal verification. J. Softw. Tools Technol. Transfer **7**(2), 156–173 (2005)
54. Prehofer, C.: Feature-oriented programming: a new way of object composition. Concurrency Comput. Pract. Experience **13**(5), 465–501 (2001)
55. Reiff, S.: Identifying resolution choices for an online feature manager. In: Proceedings of the International Workshop on Feature Interactions in Telecommunications and Software Systems, pp. 113–128. IOS Press, Amsterdam (2000)
56. Schippers, H., Molderez, T., Janssens, D.: A graph-based operational semantics for context-oriented programming. In: Proceedings of the Context-Oriented Workshop, COP 2010, vol. 6, pp. 1–6. ACM, New York (2010)
57. Schmieders, E., Metzger, A.: Preventing performance violations of service compositions using assumption-based run-time verification. In: Abramowicz, W., Llorente, I.M., Surridge, M., Zisman, A., Vayssiére, J. (eds.) ServiceWave 2011. LNCS, vol. 6994, pp. 194–205. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24755-2_19
58. Sistla, P.: Hybrid and incremental modelchecking techniques. ACM Comput. Surv. **28**(4es), December 1996
59. Song, H., Barrett, S., Clarke, A., Clarke, S.: Self-adaptation with end-user preferences: using run-time models and constraint solving. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 555–571. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41533-3_34
60. Straeten, R.V.D.: Inconsistency Management in Model-Driven Engineering. Ph.D. thesis, Vrije Universiteit Brussel, Pleinlaan 2, Brussels, Belgium, September 2005
61. Streett, R.S., Emerson, E.A.: An automata theoretic decision procedure for the propositional mu-calculus. Inf. Comput. **81**(3), 249–264 (1989)
62. Sykes, D., Magee, J., Kramer, J.: Flashmob: distributed adaptive self-assembly. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, pp. 100–109. ACM, New York (2011)
63. Taentzer, G.: AGG: a graph transformation environment for modeling and validation of software. In: Pfaltz, J.L., Nagl, M., Böhnen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 446–453. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25959-6_35
64. Tan, L.: Model-based self-adaptive embedded programs with temporal logic specifications. In: Quality Software International Conference, QSIC 2006, pp. 151–158, October 2006
65. Tesei, L., Merelli, E., Paoletti, N.: Multiple levels in self-adaptive complex systems: a state-based approach. In: Gilbert, T., Kirkilionis, M., Nicolis, G. (eds.) Proceedings of the European Conference on Complex Systems, ECCS 2012. Springer Proceedings in Complexity, pp. 1033–1050. Springer, Cham (2012), https://doi.org/10.1007/978-3-319-00395-5_124

66. Ukey, N., Niyogi, R., Milani, A., Singh, K.: A bidirectional heuristic search technique for web service composition. In: Taniar, D., Gervasi, O., Murgante, B., Pardede, E., Apduhan, B.O. (eds.) ICCSA 2010. LNCS, vol. 6019, pp. 309–320. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12189-0_27
67. Van Der Straeten, R., Jonckers, V., Mens, T.: A formal approach to model refactoring and model refinement. Softw. Syst. Model. **6**(2), 139–162 (2006)
68. Zhang, J., Cheng, B.H.: Using temporal logic to specify adaptive program semantics. J. Syst. Softw. **79**(10), 1361–1369 (2006)

Feedback Control as MAPE-K Loop in Autonomic Computing

Eric Rutten^{1(✉)}, Nicolas Marchand², and Daniel Simon^{3,4}

¹ INRIA, Grenoble, France
eric.rutten@inria.fr

² CNRS, GIPSA-lab, Grenoble, France
Nicolas.Marchand@gipsa-lab.fr

³ INRIA, Sophia Antipolis, France
daniel.simon@inria.fr

⁴ LIRMM, Montpellier, France

<https://team.inria.fr/ctrl-a/members/eric-rutten>,
<http://www.gipsa-lab.fr/~nicolas.marchand>, <http://www2.lirmm.fr/~simon>

Abstract. Computing systems are becoming more and more dynamically reconfigurable or adaptive, to be flexible w.r.t. their environment and to automate their administration. Autonomic computing proposes a general structure of feedback loop to take this into account. In this paper, we are particularly interested in approaches where this feedback loop is considered as a case of control loop where techniques stemming from Control Theory can be used to design efficient safe, and predictable controllers. This approach is emerging, with separate and dispersed effort, in different areas of the field of reconfigurable or adaptive computing, at software or architecture level. This paper surveys these approaches from the point of view of control theory techniques, continuous and discrete (supervisory), in their application to the feedback control of computing systems, and proposes detailed interpretations of feedback control loops as MAPE-K loop, illustrated with case studies.

Keywords: Autonomic managers · Administration loops
Control theory

1 Feedback Loops in Computing Systems

1.1 Adaptive and Reconfigurable Computing Systems

Computing systems are becoming more and more dynamically reconfigurable or adaptive. The motivations for this are that, on the one hand, these systems should dynamically react to changes on their environment or in their execution platform, in order to improve performance and/or energy efficiency. On the other

This work has been partially supported by CNRS under the PEPS Rupture Grant for the project API: <https://team.inria.fr/ctrl-a/members/eric-rutten/peps-api> and by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01).

hand, complex systems are too large to continue being administrated manually and must be automated, in order to avoid error-prone or slow decisions and manipulations.

This trend can be observed at very diverse levels of services and application software, middle-ware and virtual machines, operating systems, and hardware architectures. The automation of such dynamical adaptation manages various aspects such as computing and communication resources, quality of service, fault tolerance. It can concern small embedded systems like sensors networks, up to large-scale systems such as data-centers and the Cloud. For example, data-centers infrastructures have administration loops managing their computing resources, typically with energy-aware objectives in mind, and possibly involving management of the cooling system. At a lower level, FPGA-based architectures (Field-Programmable Gate Arrays) are hardware circuits that can be configured at run-time with the logics they should implement: they can be reconfigured dynamically and partially (i.e. on part of the reconfigurable surface) in response to environment or application events; such reconfiguration decisions are taken based on monitoring the system's and its environment's features.

1.2 Autonomic Computing

Autonomic computing [50] proposes a general feedback loop structure to take adaptive and reconfigurable computing into account. In this closed loop, systems are instrumented with monitors of sensors, and with reconfiguration actions or actuators; these two have to be related by a control and decision component, which implements the dynamic adaptation policy or strategy. The loop can be defined as shown in Fig. 1 with the MAPE-K approach, with sub-components for Analysis of Monitored data, Planning response actions, Execution of these actions, all of them based on a Knowledge representation of the system under administration. Autonomic computing has now gained a large audience [54].

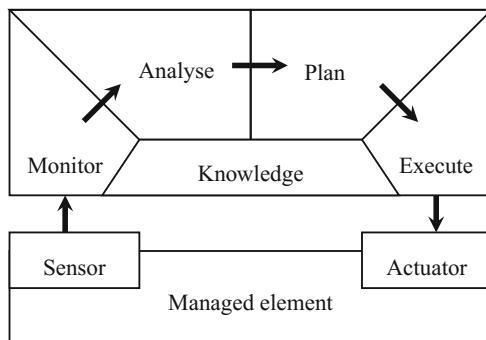


Fig. 1. The MAPE-K autonomic manager for administration loop.

Such autonomic loops can be designed and developed in many different ways, relying on techniques from e.g. Artificial Intelligence. However, an important

issues remains in that it is generally difficult to master the behavior of the automated closed-looped systems with precision.

1.3 Need for Control

We are therefore particularly interested in an approach where this MAPE-K loop is considered as a case of a control loop. Then, techniques stemming from control theory can be used to design efficient, safe, and predictable controllers. Control theory provides designers with a framework of methods and techniques to build automated systems with well-mastered behavior. A control loop involves sensors and actuators that are connected to the process or “plant” i.e., the system to be controlled. A model of the dynamical behavior of the process is built, and a specification is given for the control objective, and on these bases the control is derived. Although there are approaches to the formal derivation of software from specifications, such as the B method [3], this methodology is not usual in Computer Science, where often a solution is designed directly, and only then it is analyzed and verified formally, and the distinction between the process and its controller is not made systematically.

We observe that this approach of using Control Theory methods and techniques for computing systems, although well identified [44, 77], is still only emerging. Works are scattered in very separate and dispersed efforts, in different areas of the field of reconfigurable or adaptive computing, be it at software or architecture level, in communities not always communicating with each other. Some surveys are beginning to be offered [35], some offering a classification [66], or concentrating on Real-Time computing systems [7, 22] but a synthesis of all these efforts is still lacking. The community begins to structure itself, notably around workshops focused specifically on the topic e.g., Feedback Computing [21].

There exist related works in the community on Software Engineering for self-adaptive systems presenting approaches to integrate Control Theory, typically with continuous models [34]: here we focus on the MAPE-K loop as a target for Control techniques. Also, some works exist considering different approaches to discrete control, in the sense of considering events and states (see Sect. 3.3, [56]) related to planning techniques from Artificial Intelligence e.g., [17, 28, 70] or reactive synthesis in Formal Methods or Game Theory e.g., [13, 29]. A wide overview is given in another chapter of this book [56]. In this paper we make the choice to focus in more detail on approaches from the Control Theory community, based on continuous models and on the supervisory control of Discrete Event Systems [18, 69, 73].

Our point in this paper is to contribute to relating on the one hand, the general notion of control loop in Autonomic Computing, and on the other hand, design models and techniques stemming specifically from Control Theory. In doing so, we perform choices and do not pretend to exhaustivity. In particular, we do not include in our scope techniques from different approaches and communities like, e.g., Artificial Intelligence or Formal Methods, even though they may have features not covered here.

Indeed several layers and different flavors of control must be combined to fully handle the complexity of real systems. This already lead a long time ago to hierarchical control, e.g. [5], where low level (i.e. close to the hardware) fast control layers are further coordinated by slower higher level management layers. Besides the different bandwidth between layers, it also happens that different control technologies must be combined. For example, it is often observed in robot control architectures that low level control laws, designed in the realm of continuous control, are managed by a control actions scheduler based on discrete events systems, such as in [2]. Both the continuous and discrete time control designs described in the next sections are example of building blocks to be further used in hierarchical control for Autonomic Computing.

1.4 Outline

This paper proposes interpretations of the MAPE-K loop from Autonomic Computing in terms of models and techniques from Control Theory. We first consider this from the point of view of continuous control in Sect. 2, starting with the classical PID up to more elaborate nonlinear and event-based control.

Discrete control is then considered in Sect. 3, where discrete event systems are modeled by transition systems e.g., Petri nets or labelled automata.

Then some illustrative case studies are presented in Sect. 4, showing how these concepts can be put into practice in the framework of real-world computing systems.

Finally, Sect. 5 provides discussions and perspectives.

2 Continuous Control for Autonomic Computing

2.1 Brief Basics of Continuous Control

The basic paradigm in control is *feedback*, or *closed-loop* control, as depicted in Fig. 2. The underlying idea is that control signals are continuously computed from the *error signal*, i.e. the difference between the actual behavior of the plant (measured by its outputs) and its desired behavior (specified by a reference signal). The loop is closed through the controlled plant, and control actions are computed indefinitely at a rate fast enough with respect to the process dynamics [8, 9].

The behavior of the controlled process, whatever its nature (e.g. electro-mechanical or chemical for physical devices, but also digital for schedulers, databases and other numerical components) is never perfectly known. It is known only through a dynamic *model* which aims to capture the main characteristics of the process. It is most often given as of a set of difference equations where the continuous time is sampled (usually periodically) at instants t_k, t_{k+1}, \dots :

$$\begin{aligned} \mathbf{x}_{k+1} &= f(\mathbf{x}_k, \mathbf{u}_k), & x(t=0) &= x_0 \\ \mathbf{y}_k &= g(\mathbf{x}_k) \end{aligned} \tag{1}$$

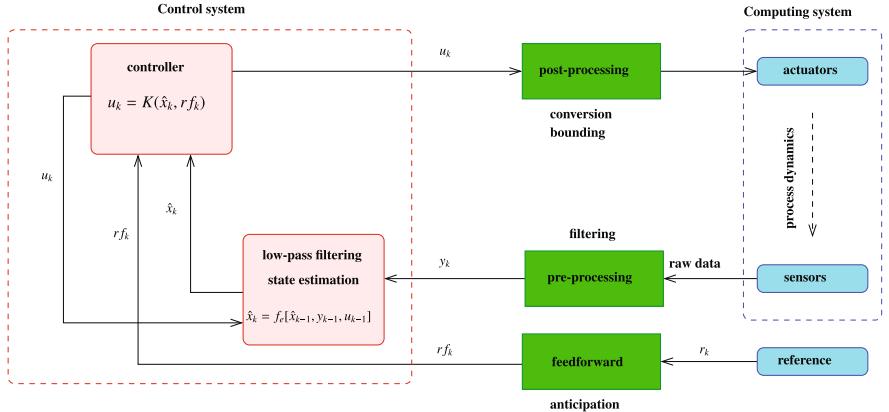


Fig. 2. The control loop for continuous control.

Here \mathbf{x} is the state vector of the process able to describe its behavior over time, x_0 is its value at the initialization of the system. \mathbf{y} is a vector of outputs measured on the process via sensors and $f(\cdot)$ and $g(\cdot)$ are respectively the state evolution function and the output function of the plant model. \mathbf{u} is the vector of control signals sent to the actuators, it is repetitively computed by a controller, e.g. by a state feedback as:

$$\mathbf{u}_k = K(\hat{\mathbf{x}}_k, \mathbf{r}_k) \quad (2)$$

where $\hat{\mathbf{x}}$ is an estimate of the state vector, \mathbf{r} is a set of reference signals to be tracked and $K(\cdot)$ is a function (which can be static or dynamic) of the state (or outputs) and of the reference signals.

For example, considering the control model of a web server, the system state may gather the number of admitted requests and an estimate of the CPU load, the observed output vector may gather a measure of the CPU activity filtered over some time window and the observed response time, and the control input can be an admission controller. The control objective can be the result of the minimization of a Quality of Service criterion gathering the rejection rate and the response time of the server under constraint of CPU saturation avoidance. Note that most often the variables of interest are not directly available from simple measurements, and that the system state must be reconstructed and smoothed using signal processing and filtering from the raw sensors outputs.

Remember that (1) is a model of the plant, i.e. an abstraction of reality where the structure is simplified and the value of the parameters is uncertain. Therefore, an open-loop control, using the inverse of the model to compute control inputs from the desired outputs, inevitably drifts away and fails after some time. Compared with open-loop control (which would rely on an utopian perfect knowledge of the plant), the closed-loop structure brings up several distinctive and attractive properties:

Stability. Briefly speaking, a system is said stable if its output and state remain bounded provided that its inputs and disturbances remain bounded (formal definitions, such as BIBO stability, Lyapunov stability and others, can be easily found thorough the control literature). It is a crucial property of a control system, ensuring that the trajectory of the controlled plant is kept close enough to the specified behavior to satisfy the end-user's requirements. A distinctive property of feedback controllers is that, if they are well designed and tuned, they are able to improve the stability of marginally stable plants, and even to stabilize naturally unstable systems, e.g., modern aircrafts. Anyway the stability of a control system must be carefully assessed, as poor design or tuning can drive the system to instability;

Robustness. The control actions are repeated at a rate which is fast compared with the system dynamics, hence the increments of tracking errors due to imperfect modeling are small, and they are corrected at every sample. Indeed, besides the main directions of the dynamics, the model does not need to capture the details of the process dynamics. Most often, poorly known high frequencies components are discarded, and exact values of the parameters are not needed. Therefore, feedback is able to provide precise and effective control for systems made of uncertain components.

Tracking and performance shaping. As the controller amplifies and adjusts the tracking error before feeding the actuators, it is able to shape the performance of the controlled plant. For example, the closed-loop system can be tuned for a response faster than the open-loop behavior, and disturbances can be rejected in specified frequency bands. Thanks to robustness, tracking objectives can be successfully performed over a large range of input trajectories without need for on-line re-tuning;

2.2 The MAPE-K Loop as a Continuous Control Loop

The MAPE-K description corresponding to the model (1) with control (2) is as shown in Fig. 3. The *Monitor* phase of the MAPE-K loop corresponds to the sampling of the system, typically, it defines the frequency at which the data must be acquired. It is usually related to sampling theory (Shannon theorem). The *Analyse* phase is in our description represented by the hat over x . This notation is commonly used to denote that the exact value of x is not known, either because of noise that requires a filtering action or because it can not be measured directly from the system. It is for instance the case of energy consumption that must be estimated using other variables like CPU or disk usage. The analyse phase would in that case include the signal estimation/reconstruction or more simply the filtering. The *Plan* phase corresponds to the computation of the control law using the Knowledge of the system held in the model. Finally, the *Execute* phase consist in changing the value of the actuator at a frequency which is most often identical to the sampling frequency of the monitor phase.

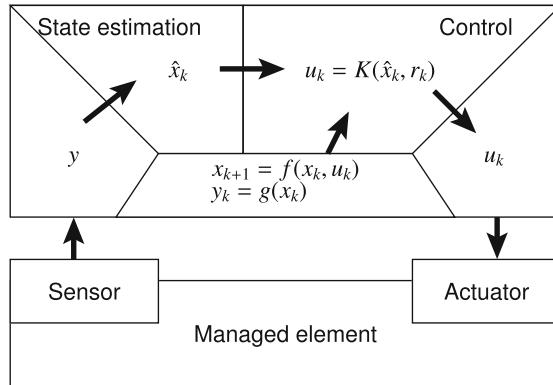


Fig. 3. The continuous control loop as a MAPEK diagram

2.3 Continuous Feedback Computing

Let us now examine how feedback can be applied to computing system administration, resource management or network management. Although feedback control was first developed to control physical devices like steam engines, factory lines or various kind of vehicles [9], the closed-loop concept has been adapted for the control of digital devices such database servers, e.g. [44, 60], or real-time schedulers as in [62]. However, compared with usual control applications, the nature of the controlled process deeply differs in the case of control for computing devices, and the usual components of the control loops must be adapted for the particular technology.

Models. Usual models for the control of continuous process are given as a set of differential equations, which are further discretized as difference equations for numerical control purpose. In contrast, at a detailed level, digital objects can be often described by large FSMs which are not well suited for closed-loop control. However, thanks to robustness, a feedback-control compliant model only needs to capture the essential of the plant dynamics. For example, computing devices can be approached by “fluid modeling” where, e.g., flow equations describe flows of input requests and levels in tanks represent the state of message queues [44]. Using such abstractions leads to quite simple difference models, where for example queues behave as integrators and provide the basic dynamics in the model. Besides metrics related to performance, as computation loads or control bandwidth, some relevant properties of a software are related to reliability. For example, the failure probabilities of software components may lead to a reliability formal model given as a Discrete Time Markov Chain, further used for the design of a predictive feedback control law [33]. Some control related modeling issues are detailed in another chapter of this book [56].

Sensors. Sensors provide raw measurements from the controlled process, they are provided by hardware or software probes in the operating system or in the

application code. Basic measurements record the system activity such as the CPU use, deadlines misses or response times of individual components. Raw measurements are gathered and pre-processed to provide compound records and quality of service clues to the controller. Note that the CPU use is always of interest, as CPU overloading is a permanent constraint. However, it is only meaningful when estimated over windows of time: the size of these measurement windows and associated damping filters provide a second main source of dynamics in the plant model.

Actuators. In software control, the actuators are provided by function calls issued by the operating system, or by other software components with enough authority. Scheduling parameters such as clock rates, deadline assignments and threads priorities can be used to manage multitasking activities. Admission control can be used to handle unpredictable input request flows. The frequency scaling capability of modern chips is also an effective tool to optimize the energy consumption due to high speed calculations in mobile devices.

Controllers. Potentially all the control algorithms found in the existing control toolbox can be adapted and used for the control of digital devices [66]. Most often, thanks to the usually simple dynamic models considered for software components, simple and cheap controllers can be effective as detailed in Sect. 2.4. Anyway some more complex control algorithms have been worked out to better handle the complexity of QoS control for computing systems (Sect. 2.5).

2.4 Basic Control

PID (Proportional, Integral, Derivative) control algorithms are widely used, as they are able to control many single input/single output (SISO) systems through a very simple modeling and tuning effort. In that case the control input u is written as a function of the error signal -that is the difference between a desired output and its measure on the real system- $e(t) = r(t) - y(t)$ as:

$$u(t) = K e(t) + \frac{K}{T_i} \int_0^t e(\tau) d\tau + K T_d \frac{d}{dt} e(t) \quad (3)$$

Here the proportional term $K \cdot e(t)$ controls the bandwidth and rising time of the control loop, the derivative term $K T_d \frac{d}{dt} e(t)$ damps the oscillations and overshoots and the integral term $\frac{K}{T_i} \int_0^t e(\tau) d\tau$ nullifies the static errors, that is the value of the error $e(t)$ when t goes to the infinite.

Indeed, the ideal continuous PID must be discretized for implementation purpose. For example, using a backward difference method (with T_s the sampling period) yields

$$u_k = u_{k-1} + K[e_k - e_{k-1}] + \frac{K \cdot T_s}{T_i} e_k + \frac{K \cdot T_d}{T_s} [e_k - 2e_{k-1} + e_{k-2}] \quad (4)$$

The MAPEK diagram of the PID controller then follows as in Fig. 4. Tuning the PID is made using the knowledge of the system. This knowledge can take

the form of a state space or transfer function model but can also reside in an empirical tuning of the parameter of the PID controller.

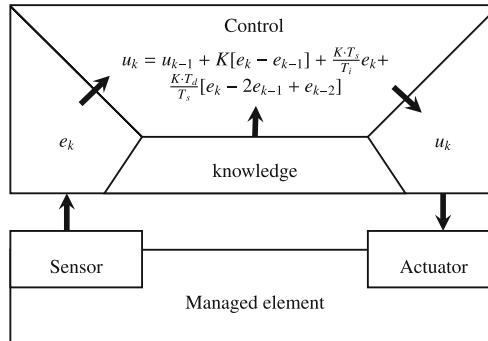


Fig. 4. The PID continuous control loop as a MAPEK diagram

2.5 Advanced Modeling and Control

Besides PID controllers which have been used in pioneering works (e.g. [59]), other simple/linear control schemes were also implemented in the context of computer science. [44] is an emblematic example of a black-box modeling in order to derive a controller using classical linear control theory aiming to maximize the efficiency of Lotus Notes. Other linear approaches were also implemented for periods rescaling [19] or to control elasticity of distributed storage in cloud environment [65]. All linear systems share the same superposition property and can be analyzed and assessed using well established mathematical tools. Unfortunately, their use to real systems that are most of the time non-linear is possible only on a limited range of the state space for which linearization is meaningful.

Indeed, many classical non-linearities of computer systems (limited range for variables, limited bandwidth, etc.) can hardly be taken into account only with linear tools [34]. In particular, optimizing a computing system operation may need to intentionally load the actuators (i.e. the CPUs) until saturation, rather than avoiding the actuators limits as in the linear control approach. Therefore, in addition to the linear control theory, the control toolbox now contains a rich set of advanced control algorithms, which have been developed over years to supplement the shortage of simple linear control in specific cases.

For example, early models of servers were simple linear fluid models and the corresponding linear controller as [44]. However, handling thrashing in servers needs to model the overhead due to parallel operations: the resources needed by a server to serve requests is not proportional to the number of requests. Non-linear models and control are needed in that case detailed in Sect. 4.2 [62].

In another case study [67], handling the static input and output non-linearities of a software reservation system is made by the combination of linear

and non-linear blocks in a Hammerstein-Wiener block structure. Then, the corresponding QoS controller is designed in the predictive control framework. Note that even when these more elaborated non-linear models are considered, the resulting controllers remain simple with a very small run-time overhead and a moderate programming effort.

Other non-linear, switched, hybrid, hierarchical and cascaded schemes were implemented on various computing systems (see for instance [76, 78] and the references therein).

Indeed it appears that, considering the quite simple dynamics of the controlled system, the time devoted to modeling is by far larger than the time devoted to control design. Models well suited for control purpose must be simple enough to allow for the synthesis of a control algorithm, while being able to capture the essence of the system behavior. Typically, modeling for the control of autonomic computing systems needs to consider trade-offs between the control objectives and the cost needed to reach them through the execution of parallel activities running on shared hardware and software resources [55].

For example, it has been shown in [61] that a game theoretic framework allows for the decoupling between the resource assignment and the quality setting, finally leading to a resource manager with a linear time complexity in the number of applications running in parallel. Once the resources and concurrent activities have been suitably modeled, the control decisions can be implemented as a hierarchy of layered controllers ranging from the control of elementary components up-to QoS optimization, e.g., as in [57].

Finally, the execution cost of the controller itself must be considered. Traditionally control systems are time triggered, allowing for a quite simple stability analysis in the framework of periodic sampling. However, the choice of the triggering period is an open issue, as reactivity needs fast sampling leading to a high computing cost. However, fast reactions are not always necessary, for example in case of slowly varying workloads. To avoid wasting the computing resource, the event-based control paradigm has been proposed (e.g. [75]). With this approach, the controller is activated only when some significant event acting on the system triggers an event detector.

3 Discrete Control for Autonomic Computing

3.1 Brief Basics of Supervisory Control of Discrete Event Systems

Amongst the different approaches to discrete control (see Sects. 1.3 and 3.3, [56]) this Section focuses on and technically details the supervisory control of Discrete Event Systems [18, 69]. Figure 5 shows a control loop for the case of discrete control with a memorized state, a transition function, and a supervisory controller obtained by discrete controller synthesis.

The characterization of Discrete Event Systems [18] is given by the nature of the state space of the considered system: when it can be described by a set of discrete values, like integers, or vectors of Booleans, and state changes are observed only at discrete points in time, then such transitions between states are

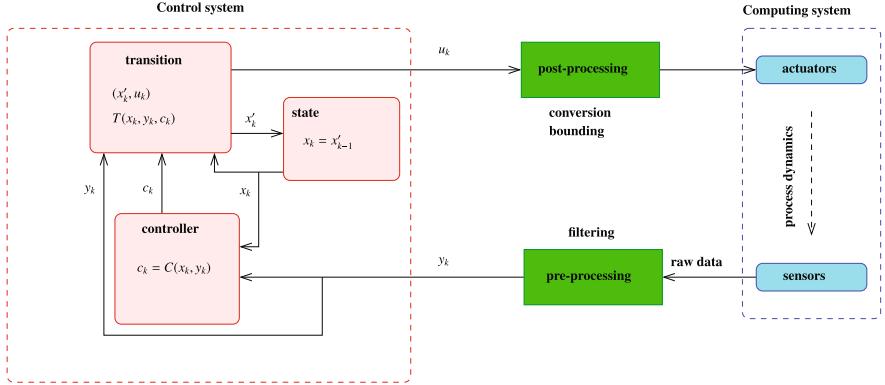


Fig. 5. The control loop for discrete control.

associated with events. In this section we very briefly and informally summarize some essential notions.

The modeling of *sequences* of such events, like the sequence of values, at time k , of y_k and u_k in Fig. 5, can be approached by *formal languages*. They enable to specify structure in the sequences of events, representing possible behaviors of a system, or desired behaviors in the interaction with a system. Operations on languages can help composing them, or making computations on the represented behaviors.

Automata are formal devices that are capable of representing languages, in the graphical and intuitive form of state and transition graphs, also called transition systems, or Finite State Machines (FSM). As shown in Fig. 5, they involve two main features. On the one hand, there is a memorizing of a *state*, the current value x_k resulting from the previous transition at $k - 1$ (with an initial value x_0 at time 0). On the other hand, is a *transition function* T computing the next value of the state x'_k in function of the current observed value y_k (we do not yet distinguish controllable variables c) and current state x_k . It can also compute values u_k that can be used to send commands to the controlled system. Figure 5 also shows possible pre-processing between raw data and y_k , or post-processing between u_k and concrete actions, e.g. corresponding to implementation-specific filters.

$$\begin{aligned} (\mathbf{x}'_k, \mathbf{u}_k) &= T(\mathbf{y}_k, \mathbf{x}_k) \\ \mathbf{x}_k &= \mathbf{x}'_{k-1} \\ \mathbf{x}_0 &= \mathbf{x}_0 \end{aligned} \tag{5}$$

Such automata can be associated with properties pertaining to their general behavior e.g., determinism, reactivity, or more specific like reachability of a state, and manipulated with operations e.g., parallel or hierarchical composition. The transitions are labelled by the events which are recognized when they are taken. Such automata-based models are precisely the basic semantic

formalism underlying reactive systems and languages [11, 43]. Related models in terms of transitions systems also include *Petri nets*, where the transitions are connecting places which are associated with tokens: the marking of the set of places by present tokens defines the state of the Petri net. The transitions can be labelled by events and their sequences define languages. The relationship with automata is given by the graph of reachable markings of the net. Analysis of such transition systems is made possible by algorithmic techniques exploring the reachable states graph in order to check typically for safety properties (e.g., using model checking techniques and tools), or concerning diagnosis of the occurrence of unobservable events from the observations on the behavior of a system.

Control of transition systems has then been defined as the problem of restricting the *uncontrolled behaviors* of a system. The latter can be described by an automaton G , control restricts its behavior so that it remains in a subset of the language of G , defined by a control objective, describing the desired behavior. The notion of supervisory control of discrete event systems has been introduced [69], which defines a supervisor that can inhibit some transitions of G , called *controllable* (controllability of a system can be partial), in such a way that, whatever the sequences of uncontrollable events, these controllable transitions can be taken in order for the desired behavior to be enforced, and the undesirable behavior avoided. Typical desired behaviors, or control objectives, in the supervisory control approach are safety properties: deadlock avoidance, or invariance of a subset of the state space (considered good). A specially interesting property of the supervisor is it should be restricting only the behaviors violating the desired objectives, or in other terms it should be *maximally permissive*. It can be noted that this important property is possible in this approach, whereas it is not considered or defined in approached dealing with more expressive goals such as liveness. As shown in Fig. 5, the resulting synthesized controller C gives values to controllable variables c , which are part of the parameters of the transition function T :

$$\begin{aligned} (\mathbf{x}'_k, \mathbf{u}_k) &= T(\mathbf{y}_k, \mathbf{c}_k, \mathbf{x}_k) \\ \mathbf{c}_k &= C(\mathbf{y}_k, \mathbf{x}_k) \\ \mathbf{x}_k &= \mathbf{x}'_{k-1} \\ \mathbf{x}_0 &= \mathbf{x}_i \end{aligned} \tag{6}$$

Tools available to users who wish to apply such automated controller synthesis techniques, adopting the approach of supervisory control for Discrete event Systems, include: TCT, based on languages models and theory [74]; Supremica, related to the manufacturing languages of the IEC standard [4]; SMACS, which achieves Controller Synthesis for Symbolic Transition Systems with partial information [47]; Sigali, which is integrated in the synchronous reactive programming environments [63], and in the compiler of the BZR language [23]. A new tool, ReaX, extends the expressivity to Discrete Controller Synthesis for infinite state systems, and treats logic-numeric properties [12].

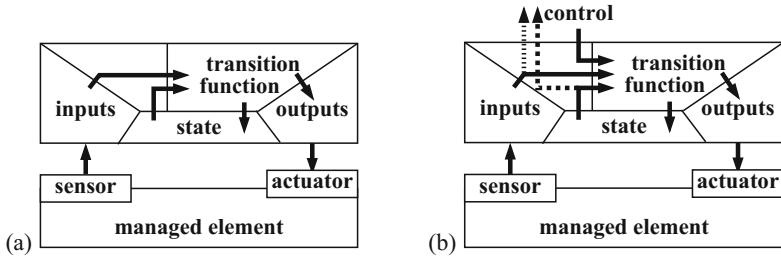


Fig. 6. The discrete control loop as a MAPEK diagram. (a): simple automaton-based manager; (b): exhibiting observability and controllability.

3.2 The MAPE-K Loop as a Discrete Supervisory Control Loop

In the general framework for autonomic computing shown in Fig. 1, discrete control can be integrated as shown in Fig. 6(a): it instantiates the general autonomic loop with knowledge on possible behaviors represented as a formal state machine, and planning and execution as the automaton transition function, with outputs triggering the actuator. As evoked in previous Section, the models used in supervisory control of DES enable to address properties on the order of events or the reachability of states, with tool-equipped techniques for verification (e.g. model checking) and especially Discrete Controller Synthesis (DCS). The latter is automated and constructive, hence we use it for the logic control of autonomic systems, encapsulated in a design process for users experts of systems, not of formalisms.

In the autonomic framework, in order to support coordination of several autonomic managers by an upper layer, some additional observability can be obtained by having additional outputs, as shown by dashed arrows in Fig. 6(b) for a FSM autonomic manager, exhibiting (some) of the knowledge and sensor information (raw, or analyzed); this can feature state information on the autonomic manager itself or of managed elements below. At that level, additional inputs can provide for controllability by an external coordinator.

3.3 Discrete Feedback Computing

As was noted by other authors, while classical control theory has been readily applied to computing systems [44], applying Discrete Control Theory to computing systems is more recent. One of the earliest works deals with controlling workflow scheduling [71]. Some focus on the use of Petri nets [45, 46, 58] or finite state automata [68].

In the area of fault-tolerant systems, some works [15, 53] present notions similar to control synthesis, not explicitly considering uncontrollables. In that sense, it resembles more open-loop control, considering the internals of a computing system, to which we prefer closed-loop control, taking into account events from its environment.

A whole line of work focuses on the computing systems problem of deadlock avoidance in shared-memory multi-threaded programs. These work rely on the literature in Discrete Control Theory concerning deadlock avoidance, which was originally motivated by instances of the problem in manufacturing systems. [73] is a programming language-level approach, that and relies upon Petri net formal models, where control logic is synthesized, in the form of additional control places in the Petri nets, in order to inhibit behaviors leading to interlocking. The Gadara project elaborates on these topics [72]. They apply Discrete Control internally to the compilation, only for deadlock avoidance, in a way independent of the application. Other works also target deadlock avoidance in computing systems with multi-thread code [10, 30].

Another kind of software problem is attacked by [37, 38]: they consider run-time exceptions raised by programs and not handled by the code. Supervisory control is used to modify programs in such a way that the un-handled exceptions will be inhibited. In terms of autonomic computing, this corresponds to a form of self-healing of the system. Applications of the Ramadge and Wonham framework to computing systems can also be found concerning component-based systems reconfiguration control, enforcing structural as well as behavioral properties [51], and more generally adaptive systems, as one of the decision techniques in a multi-tier architecture [28].

In an approach related to reactive systems and synchronous programming, discrete controller synthesis, as defined and implemented in the tool Sigali, is integrated in a programming language compiler. [27] describes “how” compilation works, with modular DCS computations, performing invariance control. This language treats expression of objectives as a first class programming language feature. The programming language, called BZR, is used in works concerning component-based software [16]. It is extended to handle logico-numeric properties, by replacing, in the modular architecture of the compiler, Sigali with the new tool ReaX [12]. Other previous work related to the synchronous languages involved some separate and partial aspects of the problem, testing the idea in the framework of a more modest specialized language [25], and particular methods and manual application of the techniques [36], and elaborating on the articulation between reactive programs and DCS [6, 24, 64], as well as application to fault-tolerance [31, 39].

As noted above, some other related work can be found in computer science and Formal Methods, in the notions of program synthesis. It consists in translating a property on inputs and outputs of a system, expressed in temporal logics, into a lower-level model, typically in terms of transition systems. For example, it is proposed as form of liberated programming [41] in a UML-related framework, with the synthesis of StateChart from Live Sequence Charts [42, 52]. Other approaches concern angelic non-determinism [14], where a non-deterministic operator is at the basis of refinement-based programming. These program synthesis approaches do not seem to have been aware of Discrete Control Theory, or reciprocally: however there seems to be a relationship between them, as well as with game theory, but it is out of the scope of this paper.

Also, interface synthesis [20] is related to Discrete Controller Synthesis. It consists in the generation of interfacing wrappers for components, to adapt them for the composition into given component assemblies w.r.t. the communication protocols between them.

4 Case Studies

4.1 Video Decoding and DVFS

Energy availability is one of the main limiting factors for mobile platforms powered by batteries. Dynamic Voltage and Frequency Scaling (DVFS) is a very effective way to decrease the energy consumption of a chip, by reducing both the clock frequency and the supply voltage of the CPUs when high computation speeds are not necessary. Many chips used in embedded or mobile systems are now fitted with such capabilities, and the computing speed is adapted on-the-fly thanks to some estimated computing performance requirement.

Using feedback loops is an effective way to robustly adapt the chip computing speed even if the incoming computation load is hard to predict, as in the example described in [32]. The problem is to minimize the energy consumption of a H.264 video decoder by using the lowest possible computing speed able to decode the frames with respect to the output display rate (25 frames/sec).

The computing speed is adapted thanks to the control architecture depicted in (Fig. 7a). At low level, a computing speed controller -integrated in silicon-drives the DVFS hardware with frequency and Vdd set points (see [32] for details). It is driven from estimates of the needed computation load (i.e. the number of CPU cycles) and decoding deadline for the incoming frame. These estimates are themselves computed by an outer frame control loop.

Measurements of decoding execution times (Fig. 7b) show that, between noisy and almost flat segments, the decoding times exhibit sharp and unpredictable isolated peaks when switching between plans. Therefore, rather than trying to compute any prediction, the estimation of the next frame computation load $\hat{\Omega}_{i+1}$ can be simply taken equal to the last one, i.e. Ω_i , recorded by the instrumentation inserted in the H.264 decoder. Even better, it can be provided by smoothed past values through a low pass filter:

$$\hat{\Omega}_{i+1} = \alpha \hat{\Omega}_{i-1} + (1 - \alpha) \Omega_i \quad (7)$$

where $0 \leq \alpha < 1$ controls the filter damping.

This rough estimate is used by the frame controller to compute the ideal deadline for the incoming frame using a simple proportional controller:

$$\Delta_{\tau_{i+1}} = \tau_{i+1} + \beta \delta_i, \quad 0 < \beta \leq 1 \quad (8)$$

where δ_i is the observed overshoot for the last decoded frame. Indeed this controller aims at driving the end-of-computation of frame $i+1$ towards τ_{i+1} , which is the theoretic timing of the periodic video rate.

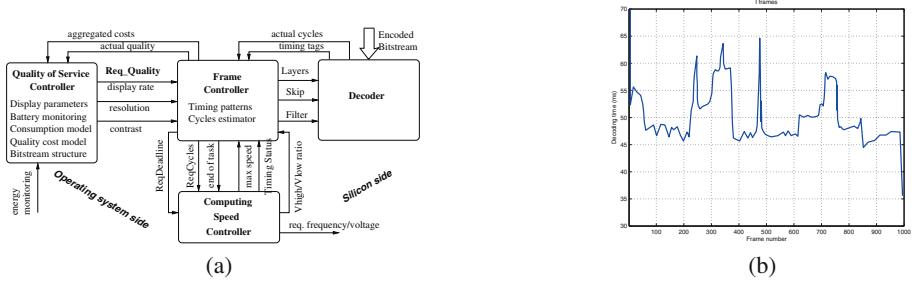


Fig. 7. (a) Control architecture – (b) Frames computation times

Despite the apparently overly simple computing load model (7), the very simple and low cost frame controller (8) is able to regulate the decoding timing overshoot to very small values (Fig. 8a), thus keeping an fluid display rate. Computing a penalty function based on the viewing quality (Fig. 8b) shows that using these elementary feedback loops allow both for a better viewing quality and up-to 24% energy saving compared with the uncontrolled decoding case.

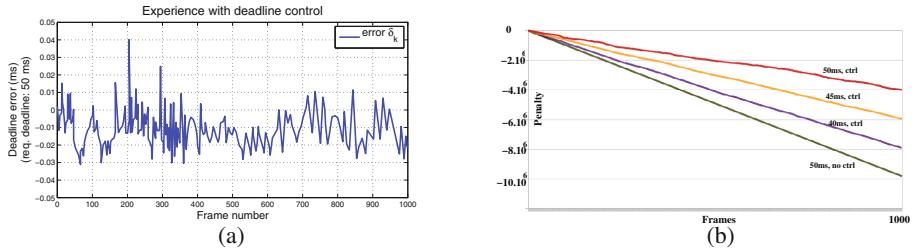


Fig. 8. (a) Deadline with control – (b) Video viewing penalty function

Hence, this example show that even very simple control loops with negligible computation overheads, if carefully designed, may have a very positive impact on an embedded system adaptiveness and robustness against a poorly modelled environment.

4.2 Server Provisioning

A classical technique used to prevent servers from thrashing when the workload increases consists in limiting client concurrency on servers using admission control. Admission control has a direct impact on server performance, availability, and quality of service (QoS). Modeling of servers and feedback control of their QoS has been one of the first application domain targeted by feedback scheduling, first using linear models [44, 60]. However, it appears that to handle trashing, the model must accurately capture the dynamics and the *nonlinear* behavior of server systems, while being simple enough to be deployed on existing systems.

Based on numerous experiments and identification, a nonlinear continuous-time control theory based on fluid approximations has been designed in [62]. It is both simple to use and able to capture the overhead due to the parallel processing of requests responsible for thrashing (Fig. 9a).

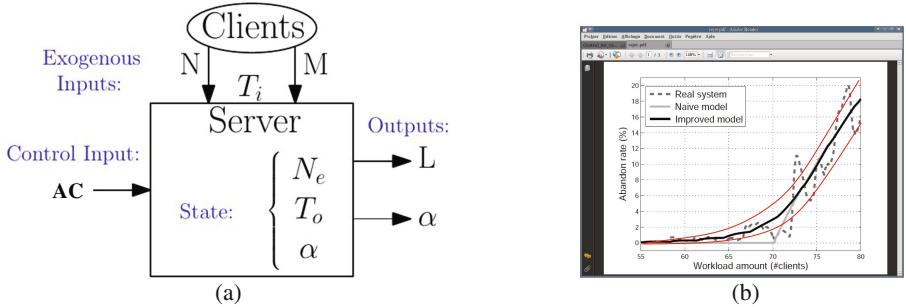


Fig. 9. (a) Fluid model – (b) Rejection rate

The request queue is considered as a fluid tank receiving client request flows M and N and emitting a served requested flow with latency L for the served requests. The system state is defined by the number of concurrently admitted requests N_e , the server throughput T_o and the rejection rate α . The modelling effort leads to the following model for the input/output latency:

$$L(N_e, M, t) = a(M, t)N_e^2 + b(M, t)N_e + c(M, t) \quad (9)$$

where the latency L is a non-linear function of the number of N_e , of the server mix load M and of continuous time t . The rejection rate is given by

$$\dot{\alpha}(t) = -\frac{1}{\Delta} \left(\alpha(t) - \frac{N_e(t)}{AC(t)} \cdot \left(1 - \frac{T_o(t)}{T_i(t)} \right) \right) \quad (10)$$

with Δ the sampling rate, T_i the input flow and AC the admission control value.

Then two control laws could be derived for different control objectives:

- $AC = \frac{N_e}{1+\gamma_L(L-L_{\max})}$ maximizes the availability of the server, i.e. minimizes the rejection rate;
- $AC = \frac{\alpha N_e}{\alpha - \gamma_\alpha(\alpha - \alpha_{\max})}$ maximizes the performance, i.e. minimizes the latency for the admitted requests.

These simple control laws are cost effective and easy to tune, as they both use a single tuning parameter γ_L or γ_α .

Despite their simplicity, using these simple controllers allows for an efficient on-line management of an Apache web server. For example, Fig. 10 show that the rejection rate can be kept close to a desired goal, or that the latency of the served requested can be regulated around the requested value.

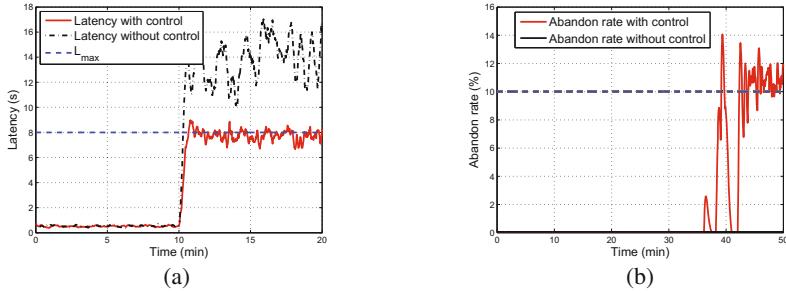


Fig. 10. (a) Latency control – (b) Rejection rate control

Even more important, these controllers –with negligible computing cost– turns the nature of the system into a safer behavior. Indeed, trashing is automatically avoided even in the case of huge overloads with no need for an operator to manually re-tune the AC parameters.

4.3 Coordination of Multiple Autonomic Administration Loops

Real autonomic systems require multiple management loops, each complex to design, and possibly of different kinds (quantitative, synchronization, involving learning, ...). However their uncoordinated co-existence leads to inconsistency or redundancy of action. Therefore we apply discrete control for the interactions of managers [40]. We validate this method on a multiple-loop multi-tier system.

Controllable managers as seen in Fig. 6(b) can be assembled in composites, where the coordination is performed in a hierarchical framework, using the possibilities offered by each of them, through control interfaces, in order to enforce a coordination policy or strategy. We base our approach on the hierarchical structure in Fig. 11: the top-level AM coordinates lower-level ones.

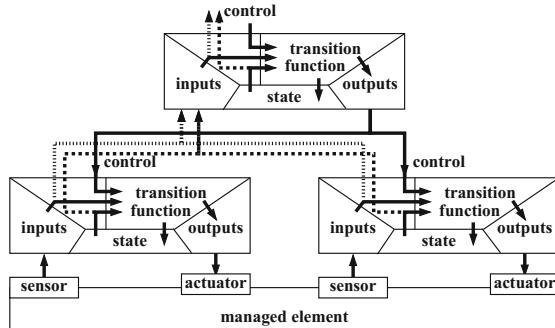


Fig. 11. Autonomic coordination for multiple administration loops.

We consider the case study of the coordination of two administration loops for the management of a replicated servers system based on the load balancing scheme. Self-sizing addresses resource optimization, and dynamically adapts the degree of replication depending on the CPU load of the machines hosting the active servers. Self-repair addresses server recovery upon fail-stop failure of a machine hosting a single or replicated server. Co-existence problems occur when failures trigger incoherent decisions by self-sizing: The failure of the load balancer can cause an under-load of the replicated servers since the latter do not receive requests until the load balancer is repaired. The failure of a replicated server can cause an overload of the remaining servers because they receive more requests due to the load balancing. A strategy to achieve an efficient resource optimization could be to (1) *avoid removing a replicated server when the load balancer fails*, and (2) *avoid adding a replicated server when one fails*.

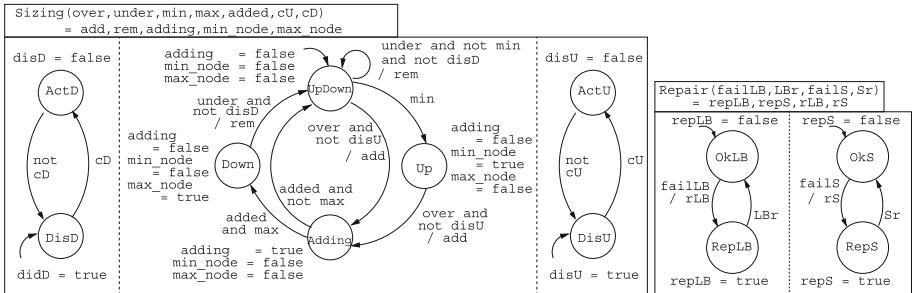


Fig. 12. Managers models: self-sizing (left) and self-repair (right).

Figure 12 shows the automata modelling the behaviors of the managers, in the BZR language [26], abstracted to the relevant activity information. In the right of the Figure, the self-sizing manager is composed of three sub-automata. In brief, the two external ones model the control of the adding (resp. removal) of servers, with $disU$ (resp. $disD$), which, when true, prevent transitions where output add (resp. rem) triggers operations. The center one models the behaviors in reaction to load variation, for which all detail is available elsewhere [40]. In the left of the Figure are self-repair managers for the load balancer (LB) and the replicated servers (S). The right automaton concerns servers, and is initially in OkS . When $fails$ is true, it emits repair order rS and goes to the $RepS$ state, where $repS$ is true. It returns back to OkS after repair termination (Sr is true). Repair of the LB is similar. The automata in Fig. 12 are composed in order to have the global behavior model, and a contract specifies the coordination policy. The policies (1) and (2) in Sect. 4.3 are enforced by *making invariant, by control* upon the controllable variables Cu , Cd , the subset of states where the predicate holds: $((repLB \Rightarrow disD) \text{ and } (repS \Rightarrow disU))$

This controller was validated experimentally on a multi-tier system based on Apache servers for load balancing (with a self-repair manager) and replicated

Tomcat servers (with both self-sizing and self-repair managers), with injection of workloads and failures to which the system responded properly, without over-reacting, according to the objective.

5 Conclusions and Perspectives

We propose a discussion of the problem of controlling autonomic computing systems, which is gaining importance due to the fact that computing systems are becoming more and more dynamically reconfigurable or adaptive, to be flexible w.r.t. their environment and to automate their administration. We observe that one approach consists of using Control Theory methods and techniques for computing systems: although it is well identified [44], it is still only emerging, and works are scattered in separate areas and communities of Computer Science.

We aim at conveying to Computer Scientists the interest and advantages of adopting a Control Theory perspective for the efficient and predictable design of autonomic systems. Compared with open-loop, closed-loop control provides adaptability and robustness, allowing for the design of fault-tolerant systems against varying and uncertain operating conditions. However, there still is a deep need for research in the problems of mapping from high-level objectives in terms of Quality of Service (QoS) or Service Level Objectives (SLO) and abstract models towards lower-levels effective actions on the managed systems. In the area of Computing Systems research, there is an important topic in the design of architectures so that they are made controllable [48], as self-aware software (adaptive, reflective, configuring, repairing...) needs explicitly built-in sensing and acting capabilities [49]. On the other side, the kind of models usual in Control Theory must be totally reworked to be useful for computing systems, and this is a research challenge for the Control community. Also, an important issue is that complex systems will involve multiple control loops, and their well-mastered composition and coordination to avoid interferences is a difficult and hardly tackled question [1].

One lesson learned in this work is that the open problems are concerning both Control Theory and Computer science, and that solutions going beyond simple cases require active cooperation between both fields. As noted by other authors e.g., [33], this bi-disciplinary field is only beginning, and the problems involved require competences in Control, as well as expertise in the computing systems. There is a costly investment in time to build common vocabulary and understanding, for which the MAPE-K loop offers a common ground, and this investment opens the way for better controlled autonomic computing systems, safer and optimized.

References

1. Abdelzaher, T.: Research challenges in feedback computing: an interdisciplinary agenda. In: 8th International Workshop on Feedback Computing, San Jose, California (2013)
2. Aboubekr, A.S., Gwenaël, D., Pissard-Gibollet, R., Rutten, É., Simon, D.: Automatic generation of discrete handlers of real-time continuous control tasks. In: IFAC World Congress 2011, Milano, Italie, August 2011
3. Abrial, J.-R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
4. Akesson, K.: Supremica. <http://www.supremica.org/>
5. Albus, J.S., Barbera, A.J., Nagel, R.N.: Theory and practice of hierarchical control. National Bureau of Standards (1980)
6. Altisen, K., Clodic, A., Maraninchi, F., Rutten, E.: Using controller-synthesis techniques to build property-enforcing layers. In: Degano, P. (ed.) ESOP 2003. LNCS, vol. 2618, pp. 174–188. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36575-3_13
7. Årzén, K.-E., Robertsson, A., Henriksson, D., Johansson, M., Hjalmarsson, H., Johansson, K.H.: Conclusions of the ARTIST2 roadmap on control of computing systems. ACM SIGBED (Special Interest Group on Embedded Systems) Rev. **3**(3), 11–20 (2006)
8. Åström, K.J., Murray, R.M.: Feedback Systems: An Introduction for Scientists and Engineers, 2nd edn. Princeton University Press, Princeton (2015). http://www.cds.caltech.edu/~murray/amwiki/index.php/Main_Page
9. Åström, K.J., Wittenmark, B.: Computer-Controlled Systems. Information and System Sciences Series, 3rd edn. Prentice Hall, Upper Saddle River (1997)
10. Auer, A., Dingel, J., Rudie, K.: Concurrency control generation for dynamic threads using discrete-event systems. In: 2009 47th Annual Allerton Conference on Communication, Control, and Computing, Allerton 2009, 30 September–2 October 2009, pp. 927–934 (2009)
11. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages twelve years later. Proc. IEEE **91**(1), 64–83 (2003). Special issue on embedded systems
12. Berthier, N., Marchand, H.: Discrete controller synthesis for infinite state systems with ReaX. In: 12th International Workshop on Discrete Event Systems, WODES 2014. IFAC, May 2014
13. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Saar, Y.: Synthesis of reactive(1) designs. J. Comput. Syst. Sci. **78**(3), 911–938 (2012)
14. Bodik, R., Chandra, S., Galenson, J., Kinelman, D., Tung, N., Barman, S., Rodarmor, C.: Programming with angelic nondeterminism. In: Principles of Programming Languages, POPL, pp. 339–352, January 2010
15. Bonakdarpour, B., Kulkarni, S.S.: On the complexity of synthesizing relaxed and graceful bounded-time 2-phase recovery. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 660–675. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05089-3_42
16. Bouhadiba, T., Sabah, Q., Delaval, G., Rutten, E.: Synchronous control of reconfiguration in fractal component-based systems: a case study. In: Proceedings of the Ninth ACM International Conference on Embedded Software, EMSOFT 2011, Taipei, Taiwan, pp. 309–318, October 2011

17. Braberman, V., D’Ippolito, N., Kramer, J., Sykes, D., Uchitel, S.: MORPH: a reference architecture for configuration and behaviour self-adaptation. In: Proceedings of the 1st International Workshop on Control Theory for Software Engineering, CTSE 2015, pp. 9–16. ACM, New York (2015)
18. Cassandras, C., Lafortune, S.: Introduction to Discrete Event Systems. Springer, New York (2008). <https://doi.org/10.1007/978-0-387-68612-7>
19. Cervin, A., Eker, J., Bernhardsson, B., Årzén, K.E.: Feedback-feedforward scheduling of control tasks. *Real-Time Syst.* **23**(1–2), 25–53 (2002)
20. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Mang, F.Y.C.: Synchronous and bidirectional component interfaces. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 414–427. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_34
21. Feedback Computing. <http://www.feedbackcomputing.org/>
22. Control for Embedded Systems Cluster: Roadmap on control of real-time computing systems. Technical report, EU/IST FP6 Artist2 NoE (2006)
23. Delaval, G.: Bzr. <http://bzr.inria.fr>
24. Delaval, G.: Modular distribution and application to discrete controller synthesis. In: International Workshop on Model-driven High-level Programming of Embedded Systems (SLA++P 2008), Budapest, Hungary, April 2008
25. Delaval, G., Rutten, E.: A domain-specific language for multi-task systems, applying discrete controller synthesis. *J. Embed. Syst.* **2007**(84192), 17 (2007)
26. Delaval, G., Rutten, É., Marchand, H.: Integrating discrete controller synthesis into a reactive programming language compiler. *Discrete Event Dyn. Syst.* **23**(4), 385–418 (2013)
27. Delaval, G., Marchand, H., Rutten, É.: Contracts for modular discrete controller synthesis. In: ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2010), Stockholm, Sweden, pp. 57–66, April 2010
28. D’Ippolito, N., Braberman, V., Kramer, J., Magee, J., Sykes, D., Uchitel, S.: Hope for the best, prepare for the worst: multi-tier control for adaptive systems. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp. 688–699. ACM, New York (2014)
29. D’ippolito, N., Braberman, V., Piterman, N., Uchitel, S.: Synthesizing nonanomalous event-based controllers for liveness goals. *ACM Trans. Softw. Eng. Methodol.* **22**(1), 9:1–9:36 (2013)
30. Dragert, C., Dingel, J., Rudie, K.: Generation of concurrency control code using discrete-event systems theory. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT 2008/FSE-16, pp. 146–157. ACM, New York (2008)
31. Dumitrescu, E., Girault, A., Marchand, H., Rutten, E.: Multicriteria optimal discrete controller synthesis for fault-tolerant tasks. In: Proceedings of the 10th IFAC International Workshop on Discrete Event Systems (WODES 2010), September 2010
32. Durand, S., Alt, A.-M., Simon, D., Marchand, N.: Energy-aware feedback control for a H.264 video decoder. *Int. J. Syst. Sci.* **46**(8), August 2013
33. Filieri, A., Ghezzi, C., Leva, A., Maggio, M.: Self-adaptive software meets control theory: a preliminary approach supporting reliability requirements. In: Proceedings of 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 283–292 (2011)
34. Filieri, A., Hoffmann, H., Maggio, M.: Automated design of self-adaptive software with control-theoretical formal guarantees. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014. ACM, New York (2014)

35. Filieri, A., Maggio, M., Angelopoulos, K., D'Ippolito, N., Gerostathopoulos, I., Hempel, A.B., Hoffmann, H., Jamshidi, P., Kalyvianaki, E., Klein, C., Krikava, F., Misailovic, S., Papadopoulos, A.V., Ray, S., Sharifloo, A.M., Shevtsov, S., Ujma, M., Vogel, T.: Software engineering meets control theory. In: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, pp. 71–82. IEEE Press, Piscataway (2015)
36. Gamatié, A., Yu, H., Delaval, G., Rutten, E.: A case study on controller synthesis for data-intensive embedded systems. In: Proceedings of the 6th IEEE International Conference on Embedded Software and Systems (ICESS 2009), HangZhou, Zhejiang, China, May 2009
37. Gaudin, B., Nixon, P.: Supervisory control for software runtime exception avoidance. In: Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering, C3S2E 2012, pp. 109–112. ACM, New York (2012)
38. Gaudin, B., Vassey, E.I., Nixon, P., Hinckey, M.: A control theory based approach for self-healing of un-handled runtime exceptions. In: Proceedings of the 8th ACM International Conference on Autonomic Computing, ICAC 2011, pp. 217–220. ACM, New York (2011)
39. Girault, A., Rutten, E.: Automating the addition of fault tolerance with discrete controller synthesis. *Int. J. Formal Methods Syst. Des.* **35**(2), 190–225 (2009). <https://doi.org/10.1007/s10703-009-0084-y>
40. Gueye, S.M.-K., de Palma, N., Rutten, E.: Coordination control of component-based autonomic administration loops. In: Proceedings of the 15th International Conference on Coordination Models and Languages, COORDINATION, 3–6 June 2013, Florence, Italy (2013)
41. Harel, D.: Can programming be liberated, period? *Computer* **41**(1), 28–37 (2008)
42. Harel, D., Kugler, H., Pnueli, A.: Synthesis revisited: generating statechart models from scenario-based requirements. In: Kreowski, H.-J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) *Formal Methods in Software and Systems Modeling*. LNCS, vol. 3393, pp. 309–324. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31847-7_18
43. Harel, D., Pnueli, A.: On the development of reactive systems. In: Apt, K.R. (eds.) *Logics and Models of Concurrent Systems*. NATO ASI Series (Series F: Computer and Systems Sciences), vol. 13, pp. 477–498. Springer, Heidelberg (1985). https://doi.org/10.1007/978-3-642-82453-1_17
44. Hellerstein, J., Diao, Y., Parekh, S., Tilbury, D.: *Feedback Control of Computing Systems*. Wiley-IEEE (2004)
45. Iordache, M., Antsaklis, P.: Concurrent program synthesis based on supervisory control. In: 2010 American Control Conference (2010)
46. Iordache, M.V., Antsaklis, P.J.: Petri nets and programming: a survey. In: Proceedings of the 2009 American Control Conference, pp. 4994–4999 (2009)
47. Kalyon, G., Gall, T.L.: SMACS. <http://www.smacs.be/>
48. Karamanolis, C., Karlsson, M., Zhu, X.: Designing controllable computer systems. In: Proceedings of the 10th Conference on Hot Topics in Operating Systems, HOTOS 2005, vol. 10, p. 9. USENIX Association, Berkeley (2005)
49. Kephart, J.: Feedback on feedback in autonomic computing systems. In: 7th International Workshop on Feedback Computing, San Jose, California (2012)
50. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. *IEEE Comput.* **36**(1), 41–50 (2003)
51. Khakpour, N., Arbab, F., Rutten, E.: Supervisory controller synthesis for safe software adaptation. In: 12th IFAC - IEEE International Workshop on Discrete Event Systems, WODES, Cachan, France, 14–16 May 2014 (2014)

52. Kugler, H., Plock, C., Pnueli, A.: Controller synthesis from LSC requirements. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 79–93. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00593-0_6
53. Kulkarni, S.S., Ebnenasir, A.: Automated synthesis of multitolerance. In: DSN 2004 Proceedings of the 2004 International Conference on Dependable Systems and Networks, pp. 209–219 (2004)
54. Lalanda, P., McCann, J.A., Diaconescu, A.: Autonomic Computing - Principles, Design and Implementation. Undergraduate Topics in Computer Science Series. Springer, London (2013)
55. Lindberg, M., Årzén, K.-E.: Feedback control of cyber-physical systems with multi resource dependencies and model uncertainties. In: 31st IEEE Real-Time Systems Symposium, San Diego, California, USA, November 2010
56. Litoiu, M., Shaw, M., Tamura, G., Villegas, N.M., Müller, H.A., Giese, H., Rouvoy, R., Rutten, E.: What can control theory teach us about assurances in self-adaptive software systems? In: de Lemos, R., et al. (eds.) Self-Adaptive Systems III. LNCS, vol. 9640, pp. 90–134. Springer, Heidelberg (2017)
57. Litoiu, M., Woodside, M., Zheng, T.: Hierarchical model-based autonomic control of software systems. ACM SIGSOFT Softw. Eng. Notes **30**(4), 1–7 (2005)
58. Liu, C., Kondratyev, A., Watanabe, Y., Desel, J., Sangiovanni-Vincentelli, A.: Schedulability analysis of petri nets based on structural properties. In: 2006 Sixth International Conference on Application of Concurrency to System Design, ACSD 2006, pp. 69–78, June 2006
59. Lu, C., Stankovic, J., Abdelzaher, T., Tao, G., Son, S., Marley, M.: Performance specifications and metrics for adaptive real-time systems. In: Real-Time Systems Symposium, December 2000
60. Lu, C., Stankovic, J.A., Son, S.H., Tao, G.: Feedback control real-time scheduling: framework, modeling and algorithms. Real-Time Syst. **23**(1/2), 85–126 (2002). Journal, Special Issue on Control-Theoretical Approaches to Real-Time Computing
61. Maggio, M., Bini, E., Chaspasis, G., Årzén, K.-E.: A game-theoretic resource manager for RT applications. In: 25th Euromicro Conference on Real-Time Systems, ECRTS13, Paris, France, July 2013
62. Malrait, L., Bouchenak, S., Marchand, N.: Experience with ConSer: a system for server control through fluid modeling. IEEE Trans. Comput. **60**(7), 951–963 (2011)
63. Marchand, H.: Sigali. <http://www.irisa.fr/vertecs/Logiciels/sigali.html>
64. Marchand, H., Bourai, P., Le Borgne, M., Le Guernic, P.: Synthesis of discrete-event controllers based on the signal environment. Discrete Event Dyn. Syst. Theor. Appl. **10**(4), 325–346 (2000)
65. Moulavi, M.A., Al-Shishtawy, A., Vlassov, V.: State-space feedback control for elastic distributed storage in a cloud environment. In: ICAS 2012, The Eighth International Conference on Autonomic and Autonomous Systems, pp. 18–27 (2012)
66. Patikirikoral, T., Colman, A., Han, J., Wang, L.: A systematic survey on the design of self-adaptive software systems using control engineering approaches. In: 2012 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Zurich, Switzerland (2012)
67. Patikirikoral, T., Wang, L., Colman, A., Han, J.: Hammerstein Wiener nonlinear model based predictive control for relative QoS performance and resource management of software systems. Control Eng. Pract. **20**, 49–61 (2012)
68. Phoha, V.V., Nadgar, A.U., Ray, A., Phoha, S.: Supervisory control of software systems. IEEE Trans. Comput. **53**(9), 1187–1199 (2004)
69. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. **25**(1), 206–230 (1987)

70. Sykes, D., Corapi, D., Magee, J., Kramer, J., Russo, A., Inoue, K.: Learning revised models for planning in adaptive systems. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE 2013, pp. 63–71. IEEE Press, Piscataway (2013)
71. Wallace, C., Jensen, P., Soparkar, N.: Supervisory control of workflow scheduling. In: Advanced Transaction Models and Architectures Workshop (ATMA), Goa, India (1996)
72. Wang, Y., Cho, H.K., Liao, H., Nazeem, A., Kelly, T., Lafortune, S., Mahlke, S., Reveliotis, S.A.: Supervisory control of software execution for failure avoidance: experience from the Gadara project. In: Proceedings of the 10th IFAC International Workshop on Discrete Event Systems (WODES 2010), September 2010
73. Wang, Y., Lafortune, S., Kelly, T., Kudlur, M., Mahlke, S.: The theory of deadlock avoidance via discrete control. In: Principles of Programming Languages, POPL, Savannah, USA, pp. 252–263 (2009)
74. Wonham, W.M.: TCT. <http://www.control.utoronto.ca/cgi-bin/dlxptct.cgi>
75. Xia, F., Tian, G., Sun, Y.: Feedback scheduling: an event-driven paradigm. ACM SIGPLAN Not. **42**(12), 7–14 (2007)
76. Yfoulis, C.A., Gounaris, A.: Honoring SLAs on cloud computing services: a control perspective. In: Proceedings of the European Control Conference (2009)
77. Zhu, X.: Application of control theory in management of virtualized data centres. In: Fifth International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks (FeBID), Paris, France (2010)
78. Zhu, X., Wang, Z., Singhal, S.: Utility-driven workload management using nested control design. In: 2006 American Control Conference, p. 6. IEEE (2006)

Reference Architectures and Platforms

An Extended Description of MORPH: A Reference Architecture for Configuration and Behaviour Self-Adaptation

Victor Braberman¹, Nicolas D’Ippolito^{1()}, Jeff Kramer², Daniel Sykes²,
and Sebastian Uchitel^{1,2}

¹ Department of Computing, Imperial College London, London, UK
ndippolito@dc.uba.ar

² CONICET and Departamento de Computación, FCEN,
Universidad de Buenos Aires, Buenos Aires, Argentina

Abstract. An architectural approach to self-adaptive systems involves runtime change of system configuration (i.e., the system’s components, their bindings and operational parameters) and behaviour update (i.e., component orchestration). The architecture should allow for both configuration and behaviour changes selected from pre-computed change strategies and for synthesised change strategies at run-time to satisfy changes in the environment, changes in the specified goals of the system or in response to failures or degradation in quality attributes, such as performance, of the system itself. Although controlling configuration and behaviour at runtime has been discussed and applied to architectural adaptation, architectures for self-adaptive systems often compound these two aspects reducing the potential for adaptability. In this work we provide an extended description of our proposal for a reference architecture that allows for coordinated yet transparent and independent adaptation of system configuration and behaviour.

1 Introduction

Self-adaptive systems are capable of altering at runtime their behaviour in response to changes in their environment, capabilities and goals. Research and practice in the field has addressed challenges of designing these systems from multiple perspectives and levels of abstraction.

It is widely recognised that an architectural approach to achieve self adaptability promises a general coarse-grained framework that can be applied across many application domains, providing an abstract mechanism in which to define runtime adaptation that can scale to large and complex systems [1].

We acknowledge financial support for this work from ANPCYT PICT 2012-0724, ANPCYT PICT 2011-1774, ANPCYT PICT 2013-2341, UBACYT 036, UBACYT 0384, CONICET PIP 11220110100596CO, MEALS 295261, PIDDEF 04/ESP/15/BAA.

Architecture-based adaptation involves runtime change of system configuration (e.g., the system's components, their bindings, and operational parameters) and behaviour update (e.g., component orchestration). A prevalent approach is to implement a framework, typically along the lines of a MAPE-K loop [2], in which upon monitoring a managed system and analysing its behaviour, a plan is selected amongst a fixed set of pre-defined adaptation strategies (e.g., [3–6]). An alternative approach is to endow the adaptive system with automated planning [7] or synthesis [8] capabilities which allow the system to develop new strategies at runtime (e.g., [9–12]).

Existing approaches to architectural adaptation (e.g. [3, 9, 13–17]) incorporate elements from two key areas to enable runtime adaptation: Dynamic reconfiguration [1, 4, 5, 18–22] and discrete-event control theory [11, 12, 21, 23–29]. The first, key for adapting the system configuration, studies how to change component structure and operational parameters ensuring that on-going operation is not disrupted and/or non-functional aspects of the system are improved. The second, key for adapting behaviour, studies how to direct the behaviour of a system in order to ensure high-level (i.e., business, mission) goals.

Although the notions of configuration and behaviour control are discussed and applied by many authors, they are typically compounded when architectures for adaptation are presented, reducing overall architectural adaptability. Automated change of configuration and behaviour addresses different kinds of adaptation scenarios each of which should be managed as independently as possible from the other. Nonetheless, configuration and behaviour are related and it is not always possible to change one without changing the other. The need for both capabilities of independent yet coordinated adaptation of behaviour and configuration requires an extensible architectural framework that makes explicit how different kinds of adaptation occur.

Consider a UAV on a mission to search for and analyse samples. A failure of its GPS component may trigger a reconfiguration aiming at providing location by triangulating over alternative sensor data. The strategy may involve passivating the navigation system, unloading the GPS component and loading components for other sensors in addition to the component that resolves the triangulation. A behaviour strategy that is keeping track of the mission status (e.g. tracking areas remaining to be traversed, samples collected, etc.) should be oblivious to this change.

A reconfiguration adaptation strategy that can cope with the GPS failure can be computed automatically at runtime using approaches based on, for example, SMT solvers or planners [10, 11] that consider the structural constraints provided in the system specification (e.g., the need for a location service), requirements and capabilities of component types (e.g., the requirements of a triangulation service) and runtime information of available component instances (e.g., the availability of other sensors).

The arrival of the UAV at an unexpected location due to, say, unanticipated weather conditions may make the current search and collection strategy inadequate. For instance, the new location may be farther away from the base than

expected and the remaining battery charge may be insufficient to allow visiting the remaining unsearched locations before returning to base. In this situation the behaviour strategy would have to be revised to relinquish the goal of searching the complete area before returning to base in favour of the safety requirement that battery levels never go below a given threshold. The new behaviour strategy may prioritise remaining areas to be searched (in terms of importance and convenience), visiting only a subset of the remaining locations as it moves towards the base station for recharging. Once recharged, the strategy may opt to attempt to revisit the entire area under surveillance but prioritising the locations previously discarded. Such behavioural adaptation should be independent to the infrastructure supporting reconfiguration control.

A behaviour strategy that can deal with unexpected deviations in the UAV's navigation plan can be computed automatically at runtime using approaches based on, for instance, controller synthesis [12] that consider a behaviour model describing the capabilities of the UAV (e.g. autonomy), environment (e.g., map with locations of interest and obstacles) and system goals (e.g. UAV safety requirements and search and analyse – liveness – requirements). Indeed, our proposal of an explicit separation of reconfiguration and behaviour strategy computation and enactment is in line with the design principles of a separation of concerns and information hiding [30]. The behaviour strategy is oblivious to the implementation that provides the services it calls, while the reconfiguration strategy supports the injection of the dependencies that are required by the behaviour strategy oblivious to the particular ordering of calls that the behaviour strategy will make. In a sense, the design principle which is known to support changeability supports runtime changeability, which ultimately is what adaptation is about.

Configuration and behaviour adaptation may however need to be executed in concert. Consider the scenario in which the gripper of the UAV's arm that is to be used to pick up samples becomes unresponsive. With a broken gripper the original search and analyse mission is unachievable. This should trigger an adaptation to a degraded goal that aims to analyse samples via on-board sensors and remote processing. This goal requires a different behaviour strategy (e.g. circling samples once found to perform a 360° analysis) but also a different set of services provided by different components (e.g. infra-red camera). Not only are both behaviour and configuration adaptation required, but also their enactment requires a non-trivial degree of provisioning: To set up the infra-red camera, the UAV requires folding the arm to avoid obstructing the camera's view; performing such an operation while in the air is risky. Hence, coordination between configuration and behaviour adaptation is needed: First, a safe landing location must be found, then arm folding must be completed, and only then can the reconfiguration start. New components are loaded and activated, and finally, a strategy for in-situ analysis, rather than analysis at the base, can start.

It is in the combined configuration and behaviour adaptation where the need for both separation of concerns and explicit architectural representation of coordination becomes most evident. Approaches to automated computation of

configuration and behaviour adaptation strategies require different input information and utilise different reasoning techniques. Both automated reasoning forms are of significant computational complexity and require careful abstraction of elidable information. Keeping resolution of configuration and behaviour adaptation separately allows reuse of existing and future developments in the fields of dynamic reconfiguration and control theory and also helps keep computational complexity low.

The broken UAV gripper scenario requires a coordinated behaviour and reconfiguration adaptation strategy. The adaptation required can be decomposed into a behaviour control problem that assumes that a reconfiguration service is available and a reconfiguration problem. The resulting behaviour strategy will be computed on the assumption that the UAV's capabilities will conform to the current configuration (e.g. grip command fails) until a reconfigure command is executed, and that from then on different capabilities will be available (e.g. infrared camera getPicture command available). The behaviour strategy computation will also consider restrictions on when the reconfigure command is allowed (e.g. when arm is folded) and new goals (360° picture analysis rather than collect). The computation of the reconfiguration strategy does not entail additional complexity and is oblivious to the fact that a behaviour strategy that involves a reconfiguration halfway through is being computed.

In the above scenario, what needs to be resolved at the architectural level of the self-adaptation infrastructure is which architectural element is responsible for the decomposition of the adaptation strategy into a behaviour strategy and a reconfiguration strategy, and also how strategy enactment is performed to allow the behaviour strategy to command reconfiguration at an appropriate time (and possibly even account for reconfiguration failure). Indeed, an appropriate architectural solution to this would enable guaranteeing that given a correct decomposition of the overall composite adaptation problem into configuration and behaviour adaptation problems, and given correct-by-construction configuration and behavioural strategies for these problems, the overall adaptation problem is correct.

In this paper we provide an extended description of MORPH, a reference architecture for behaviour and configuration self-adaptation. MORPH makes the distinction between dynamic reconfiguration and behaviour adaptation explicit by putting them as first class entities. Thus, MORPH allows both independent reconfiguration and behaviour adaptation building on the extensive work developed but also allowing coordinated configuration and behavioural adaptation to accommodate for complex self-adaptation scenarios. This paper extends and corrects the preliminary presentation of MORPH in [31], and includes an end-to-end scenario that better explains how the architecture works in addition to a substantial comparison with three existing architectures for adaptive systems.

2 MORPH: A Reference Architecture for Configuration and Behaviour Self-Adaptation

We start with a very brief introduction of the main architectural elements to give a general picture of how the architecture works before we go into detail of the workings and rationale of each element. A graphical representation of the architecture can be found in Fig. 1. In the remaining text, when we want to emphasise traceability to the figure we will use an *alternative font*, we also use *italics* to refer to our running UAV example.

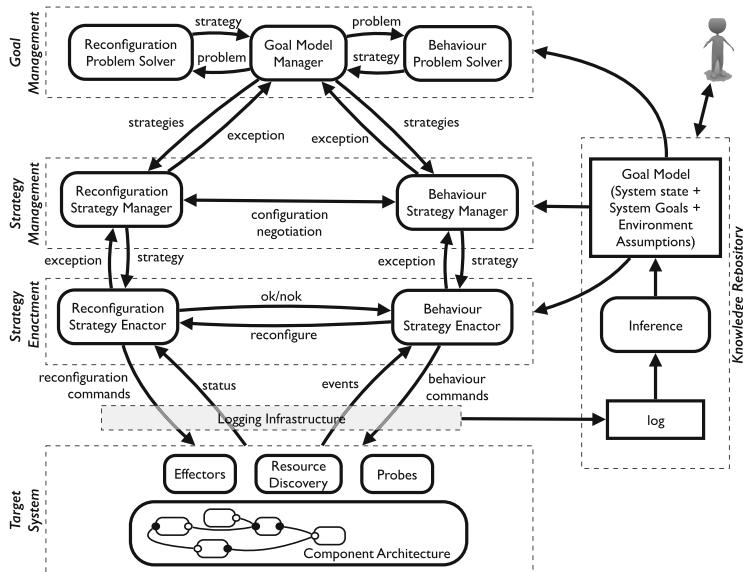


Fig. 1. The MORPH reference architecture. Arrows indicate data flow.

The architecture is structured in three main layers that sit above the target system: Goal Management, Strategy Management and Strategy Enactment. Orthogonal to the three layers is the Common Knowledge Repository. Each layer can be thought of as implementing a MAPE-K loop. The Goal Management's MAPE-K loop is responsible for reacting to changes in the goal model that require complex computation of strategic, possibly configuration and behavioural, adaptation. Its knowledge base is the Common Knowledge Repository and uses synthesis algorithms to produce at runtime strategies that can achieve system goals. The Strategy Management layer's MAPE-K loop is responsible for storing synthesised strategies and adapting to changes that can be addressed using stored strategies. It selects runtime strategies based on the Common Repository Knowledge and a set of internally managed strategies. The Strategy Enactment layer's MAPE-K loop is responsible for executing strategies; its knowledge base is primarily the strategy under enactment.

The Target System abstracts the Component Architecture that provides system functionality. The Component Architecture is harnessed by effectors and probes which allow the Strategy Enactment Layer to interface with system components. The Knowledge Repository stores in a Log the execution data produced by Target System and also stores in the Goal Model) the result of Inference procedures that produce knowledge regarding the system state, goals and environment assumptions. We expect users, administrators and other stakeholders to also produce modifications to the Knowledge Repository, and in particular the goals and environment assumptions.

The three layers that provide the architectural adaptation infrastructure are each split into reconfiguration and behaviour aspects. The Goal Management layer has a Goal Model Manager whose main responsibility is to decompose adaptation problems into reconfiguration and behaviour problems, each of which is given to a specific Solver to produce a strategy that can achieve the required adaptation. The top layer sends reconfiguration and behaviour strategies downwards. The bottom two layers have architectural elements to handle reconfiguration and behaviour strategies separately but interact with each other when and if needed to maintain overall consistency. The Strategy Management layer entities interact to ensure that they select consistent strategies to be executed by the Strategy Enactment layer (c.f., configuration negotiation). The Strategy Enactment layer entities interact to ensure that the execution of their respective strategies is done consistently over time (c.f., reconfigure command).

2.1 Target System

Responsibility: The Target System's responsibility is to achieve the system goals, encapsulate implementation details and provide abstract monitoring and control mechanisms over which the behaviour and structure of the system can be adapted.

Rationale: The rationale for this subsystem is to encapsulate the instrumentation of the system-to-be-adapted to support a flexible and reusable framework for monitoring, analysing, planning and executing adaptation strategies in the layers above.

Structure and Behaviour: The Target System, strongly inspired by [3], contains the component architecture that provides the managed system's functionality (*e.g.*, *GPS, video, telemetry and navigation components*). It also contains instrumentation to monitoring and control of the component architecture. Two types of effectors are provided. The first provides an API to add, remove and bind components, in addition to setting operational parameters of these components. We refer to the invocation of operations on these effectors as reconfiguration commands. These effectors are application domain independent, and they provide the adaptation infrastructure an abstraction over the concrete deployment infrastructure on which the component architecture runs (*e.g.*, *the UAVs operating system*). The second effector type, behaviour actions, is domain dependent and provides

an API that invokes functional services provided by components of the component architecture. *The UAV's navigation component may exhibit a complex API which is abstracted into simple commands (e.g. `goto(Location)`) that are to be used as the basis for behaviour strategies.*

The mechanism for monitoring of the component architecture can be provided by **probes** that reveal state information. As with effectors, monitoring information can be classified into two kinds. We have on one hand information regarding the status of components. This kind of information is application independent. Status of a component may indicate if it is active, inactive, connected or killed as in [10] or indicate its modes of operation as in [32]. On the other hand we refer to as events the application domain relevant information that flows from the Target System to the Strategy Enactment Layer. *UAV events may include notifications regarding battery depletion, or acknowledgements of having reached a requested location.*

Between the target system and the adaptation infrastructure a translation layer is required to provide translation services that aim to bridge the abstraction gap between the knowledge representation required to perform adaptation at the architectural level and the concrete information of the actual implementation. *In the UAV, this may include resolving event handlers, process ids, in addition to domain specific translations such the conversion of continuous variable for battery level to a discrete battery depleted event.*

Note that although in Fig. 1 there are no connecting lines between the inner components of the TargetSystem, the interaction between these components is relevant and can be far from trivial. Indeed, work on Rainbow [3], Backbone [33, 34] and by others (e.g.,[22]) is a source the reader can turn to on how such interactions can be governed.

2.2 Common Knowledge Repository

Responsibility: The key responsibility of the repository is to keep an up to date goal model at runtime based on inferences made over continuous monitoring of the environment to detect changes in goals, behaviour assumptions and available infrastructure. It is a primary interface for human-in-the-loop adaptation allowing direct manipulation of the goal model by humans.

Rationale: The design rationale for the repository is to decouple the accumulation of runtime information of the target system from the complex computational processes involved in abstracting and inferring high-level knowledge that can be incorporated, for subsequent adaptation, into a structured body of knowledge regarding stakeholder goals, environmental assumptions and target system capabilities. Note that this abstraction and inference process can range from fully automatic inference procedures to manual updates performed by humans.

Structure and Behaviour: The common knowledge repository stores information about the target system, the system goals and environment assumptions. It consists of two data structures (a log and a goal model), an Inference procedure

and an interface through which humans can inspect the log and goal model and also modify the goal model.

Information about the target system refers to historical and current information about the behaviour of the system being adapted by the three-tier adaptation infrastructure. This information includes a low level log of the evolution of the state of system components (*e.g., minute by minute recording of battery level*) and messages between them (*e.g., GPS location request by navigation component*) to aggregate data, possibly computed through complex inference procedures, including statistical information related to reliability or performance (*e.g., battery consumption ratio and predicted depletion time*).

The Goal Model: This is the key data architectural element of the repository. We use the term “goal model” in the sense of goal oriented requirements engineering [35,36] inspired on the world-machine model for requirements engineering [37,38] where the focus is on structuring information about the purpose of the socio-technical system and how such objectives can be achieved in different ways based on combinations of assumptions on the environment and requirements on the software. The purpose is sometimes referred to as high-level or business goals.

Goal models structure goals (state-based assertions intended to be satisfied over time) into refinement structures. An AND-refinement structure describes how goals can be achieved by achieving all their subgoals (*e.g., search and analyse can be achieved discretising the area into 100 m² regions, visiting each one, using image recognition to identify samples, fine grained navigation to land next to them and an arm to pick them up and take them to base*). An OR-refinement structure encodes alternative ways in which a goal may be achieved (*e.g., rather than picking samples and taking them to base, they can be analysed in-situ with an infra-red camera*). The refinement structures define an acyclic directed graph where leaf nodes need to be assigned to an entity that will guarantee them. These entities can part of the software (i.e., components) or the environment (i.e., users, external systems or laws of nature). Nodes assigned to the environment form domain assumptions (*e.g., the shape and density of samples is such that the gripper will firmly hold them and allow them to be picked up*). Goals assigned to software components are requirements (*e.g., commanding the gripper to close will make fingers clasp small objects within a 5 cm radius of its palm*).

If an AND-refinement of a top level goal is correct and all leaf nodes of the refinement are either valid assumptions or requirements guaranteed by the components they are assigned to, then the top level goal is guaranteed. OR-refinements represent alternative strategies (AND-refinements) for achieving top level goals. The selection of a specific or-refinement can depend on soft goals indicating stakeholder preferences. These preferences can be based, for instance, on non-functional requirements (*e.g., preferring a more precise location service for the UAV, preferring sample analysis to be conducted at base rather than in-situ*). Soft goals also form part of the goal model and are amenable to be structured into refinement structures.

The point of keeping a structured view of the world that includes requirements and assumptions, with multiple ways of achieving high-level goals and preference criteria over these alternatives is that at runtime it is possible to change the way a goal is achieved by selecting a different OR-refinement. The combinatorial explosion of possible OR-refinement resolutions can be a rich source for adaptation which is exploited in the Goal Management Layer. A change of the preferred OR-refinement may be triggered because the component assigned to a leaf goal of the current AND-refinement becomes available (*e.g., previously unavailable GPS component becomes functional*), or that an assumption of the AND-refinement is found to be invalid (*e.g., actual energy consumption indicates that battery power will be insufficient to complete mission*), or that the requirement that the software component was supposed to satisfy is discovered not to be provided adequately (*e.g., GPS location service behaving erratically*). In addition, this representation of rationale is amenable to being updated and changed automatically as new information is acquired. *Thus a preference between OR-refinements based on precision of the location service may be resolved differently as runtime information of their performance is accumulated.* Beyond non-functional preferences, an OR-refinement and associated preference criteria may encode different levels of functional service providing a specification of possible functionally degraded systems (*e.g., if battery consumption is a problem, an acceptable degradation may be to leave interspersed unsearched regions*).

Note that we do not prescribe a particular representation of the domain knowledge for this repository. For instance, architectural description languages (*e.g., [39, 40]*) can be used for expressing structural requirements, tabular formats (*e.g., [41]*) or contract-based specifications for describing behaviour requirements, and automata based languages (*e.g., [42]*) for expressing environment behaviour assumptions. The selection of the representation language should also consider the kinds of manual updates that may be required in addition to the expertise and preference of the humans that will perform these updates. In this paper, we have used the terminology of goal oriented requirements engineering as a way of conceptualising the information about assumptions, requirements and system objectives and state that is necessary to reason about self-adaptation.

The Inference Component: This component must be capable of abstracting runtime information accumulated in the system log and inferring high-level knowledge that can be incorporated, for subsequent adaptation, into the goal model. This component in MORPH is a placeholder for techniques (*e.g., [43]*) that build on a vast body of work in the field of machine learning covering unsupervised and reinforcement learning and also supervised learning in which a user may interact with the inference component acting as a teacher.

Human-in-the-Loop: Although not part of the MORPH knowledge repository, humans can play a central role in system adaptation and the knowledge repository is the interface that should allow them to intervene the system setting new system goals, changing the assumptions that the system can rely on. One way of viewing the human-in-the-loop, is as a stakeholder that understand high-level

goals (e.g., business goals) and assumptions (e.g., regulations) and that introduces them into the knowledge repository (possibly by refining them through some requirements engineering process). Another way of viewing the human-in-the-loop, is as human complement to the computerised inference component, plus effectors and probes, with the advantages of human intelligence, perception and motor skills, and also the disadvantages of human behaviour such as error-proneness, ulterior motives, etc. [44, 45]).

2.3 Goal Management Layer

Responsibility: The main responsibility of the Goal Management Layer is to pro-actively synthesis strategies for different combinations of goals, assumptions and system capabilities as knowledge in the Knowledge Repository evolves. Such adaptation strategies must be decomposed into behaviour and reconfiguration strategies.

Rationale: The rationale for this layer is based on three core concepts: The first is that the adaptive system must be able to compute at runtime new strategies and not be limited to a set of design-time computed tactics. This capability is sometimes referred to meta-adaptation. The second is that the adaptive system must be capable of performing strategic, computationally expensive, planning independently of and concurrently with the execution of strategies currently in place. The third is to decompose adaptation into a behaviour strategy that controls the system to an interface and a reconfiguration strategy that injects the dependencies on concrete implementations that the behaviour strategy will use. Decomposing adaptation along the modular design improves support for adaptability allowing behaviour and configuration changes independently.

Structure and Behaviour: The layer has three main entities, the Goal Model Manager, the Behaviour Problem Solver and the Reconfiguration Problem Solver.

The Goal Model Manager: This is the key element of the layer, and is responsible for three core tasks: The first is to decide when a new adaptation strategy must be computed, the second is to resolve all OR-refinements in the goal model and select the requirements to be achieved by the system, and third is to decompose the requirements into achievable reconfiguration and behaviour problems. The concrete strategies for reconfiguration and behaviour are computed by the solvers.

Production of adaptation strategies can be triggered by requests for plans from layers below or internally due to the identification of significant changes in the goal model. The former case corresponds to a scenario in which a failure is propagated rapidly upwards from the target system: *For instance, the UAV's gripper component fails. The Strategy Enactment layer, which is executing a strategy that requires the gripper, immediately declares that its current strategy is unviable and requests a new strategy from the Strategy Management Layer. However, all computed plans in the middle layer are based on having some form*

of arm to pick up objects to be studied. Consequently, the computation of a new strategy for achieving system goals is requested to the Goal Model Manager.

The alternative, internal, triggering mechanism corresponds to scenarios in which the goal model is changed because of new information inferred from the log or input manually by some stakeholder. *For instance, weather conditions may lead to inferring higher energy consumption rates from logged information. What would follow is a revision of the assumptions on UAV autonomy stored in the Goal Model. Such alteration may trigger the re-computation and downstream propagation of search strategies to make more frequent recharging stops.*

The selection based on soft goals of a specific set of requirements (leaf nodes assigned to the target system), resulting from a particular or-refinement resolution, requires information about the current system state and possibly aggregate information on its past execution (*as with energy consumption*). *For instance, with no components capable of picking up samples for transportation, an OR-refinement representing degraded levels of service (in which samples are inspected in-situ rather than at base) will only be viable. On the other hand, with a gripper component functioning, a preference on the quality of sample inspection will lead to selecting strategies that perform base-located inspections. Another example may be the selection of the position system used (GPS vs. hybrid positioning) based on component availability and precision.*

Adaptation strategies are decomposed into a strategy for achieving the component configuration that can provide the functional services required to achieve selected requirements and a behaviour strategy that can call these services in an appropriate temporal order to satisfy the requirements. *The adaptation strategy that deals with the broken gripper must reconfigure the system to use a different set of components (e.g. the infra-red camera) and coordinate its use upon reaching a position where there is a sample to be inspected.*

As discussed in the Introduction section, decomposition allows adaptation of the system configuration transparently to the behaviour strategy being executed (*e.g., changing the location mechanism*) or the behaviour strategy transparently to the configuration in use (*e.g., changing the route planning strategy*). In addition, decomposition allows the computation of multiple behaviour strategies for a given configuration (*e.g. different search and collect strategies that assume different UAV autonomy can be run on a configuration that has a gripper component*) and different configurations can be used for a given behaviour strategy (*e.g. different configurations for providing a positioning service can be used for the same search strategy*).

One of the design rationales for this layer is the runtime pre-computation of expensive adaptation strategies that are then ready to use when needed. This means that multiple reconfiguration and behaviour strategies may be constructed. Indeed, the Goal Model Manager can pre-compute, and propagate downwards at runtime many reconfiguration and behaviour strategies for one resolution of the OR-refinements of the goal model. *This may be useful, for example, if it is known that information regarding UAV autonomy is imprecise, multiple (behaviour) search strategies for searching the area may be developed so*

*that the infrastructure can adapt quickly as soon as the predicted UAV autonomy differs significantly from what can be inferred from the monitored energy consumption. Similarly, should the GPS-based location service be known to fail (perhaps do to environmental conditions), then various reconfiguration plans may be pre-computed at runtime to allow adaptation to alternative positioning systems when needed. Furthermore, adaptation strategies for different resolutions of the goal model's OR-refinements may be pre-computed at runtime. For instance, upon detecting an unreasonably high failure rate of the gripper component attempting to pick samples up, the goal model manager may produce a strategy for a degraded system objective (*inspect in-situ*) that does not require a gripper component. Should the gripper finally fail completely (or its failure rate become unacceptably low) then the pre-computed strategy can be put in place rapidly.*

Runtime pre-computation of multiple reconfiguration and behaviour strategies for different OR-refinement resolutions requires keeping additional consistency information. A many to many consistency relation must be maintained by the data structures representing reconfiguration and behaviour strategies so to allow lower levels to ensure consistent selection of a strategy of each kind. Furthermore, a relation between these strategies and the OR-refinement resolutions that they have been designed for must also be kept and propagated downstream to lower layers.

Configuration Problem Solver: The layer has two entities capable of automatically constructing strategies for given adaptation problems. The Configuration Problem Solver focuses on how to control the target system to achieve a specified configuration given the current system configuration, configuration invariants that must be preserved and component availability. The configuration to be reached may be partially specified. *For instance, the target configuration may be required to have a component that provides a positioning service. Ensuring that a GPS component is instantiated in the target system could satisfy such a requirement. However, a component using a hybrid positioning system may require additional components to be present (e.g. WI-FI, Bluetooth, Mobile phone) and bound to it.* Configuration invariants may include structural restrictions forcing the architecture to conform to some architectural style or other considerations based, for instance, on non-functional requirements. *In the UAV such restrictions may include that the attitude control components never be disabled or that the total number of active components never be beyond a given threshold to avoid battery overconsumption.*

Reconfiguration problem solvers build strategies that call actions that add and remove components, activate and passivate them, and establish or destroy bindings between them. These reconfiguration commands are part of the API exposed by the target system. The strategy may sequence these actions or have an elaborate scheme that decides which actions to call depending on feedback obtained through the information on the status of components exhibited by the target system API.

To automatically construct strategies the solvers can build upon a large body of work developed in the Artificial Intelligence and Verification communities, including automatic planners (e.g. [46]), controller synthesis (e.g. [26]), and model checking (e.g. [47]). Such techniques have been applied to construction of reconfiguration strategies in [9–11]

Behaviour Problem Solver: This entity focuses on how to control the target system to satisfy a behaviour goal. In contrast to reconfiguration problems, the behaviour goal may not be restricted to safety and reachability (i.e. reach a specific global state while preserving some invariant). Behaviour goals may include complex liveness goals such as to have the UAV monitor indefinitely an area for samples to inspect. Behaviour problem solvers produce strategies, which can be encoded as automata that monitor target system events and invoke target system actions.

In addition to the expressiveness of goals that behaviour strategies must resolve, there is an asymmetry between reconfiguration and behaviour problems. To resolve the coordination problem between strategies (*as with folding the UAV arm before a reconfiguration to deal with a gripper failure can be executed, see Introduction*), the behaviour strategies produced by the solver can invoke a reconfigure command, which triggers the execution of a reconfiguration strategy (see Sect. 2.4).

Techniques that build automatically behaviour strategies are typically based on planning and controller synthesis techniques and have been used in approaches such as [23, 43, 48].

2.4 Strategy Management Layer

Responsibility: This layer’s main responsibility is to select and propagate pre-computed behaviour and reconfiguration strategies to be enacted in the layer below. For this, the layer must store and manage pre-computed behaviour and reconfiguration plans, and request new strategies to the layer above when needed. It must also ensure that the behaviour and reconfiguration strategies sent to the lower layer are consistent, indicating their relationships.

Rationale: The main concept for the layer is to allow rapid adaptation to failed strategy executions (or capitalising rapidly on opportunities offered by new environmental conditions) by having a restricted universe of pre-computed alternative behaviour and reconfiguration strategies that can be deployed independently or in a coordinated fashion.

Structure and Behaviour: The layer has two entities that work in similar fashion mimicking much of the layer’s responsibilities but only on either behaviour or reconfiguration strategies. However, the Behaviour Strategy Manager and the Reconfiguration Strategy Manager are not strictly peers. In some adaptation scenarios the former will take a Master role in a Master-Slave decision pattern.

Behaviour Strategy Manager: The manager stores multiple behaviour strategies from which it picks one to be enacted in the layer below. The selection of strategy

may be triggered by an exception raised by the layer below or internally due to a change identified in the common knowledge repository. The former may occur when the behaviour strategy being executed finds itself in a unexpected situation it cannot handle. *For instance, the UAV executing a particular search strategy expects to be at a specific location with at least 50% of its battery remaining but finds that it is below that threshold, invalidating the rest of the strategy for covering the area to be searched.* At this point the Strategy Enactment Layer signals that the assumptions for its current strategy are invalid and requests a new strategy to this layer.

The other scenario that can trigger the selection of a new strategy is a change in the common knowledge repository. *Consider again the problem of unexpected energy overconsumption. An inference process in the knowledge repository may update the average energy consumption rate periodically based on Target System information being logged. This average may be well above the assumed consumption average for the behaviour strategy being executed. The Behaviour Strategy Manager may decide that it is plausible that the current behaviour strategy will fail and may decide to deploy a more conservative search strategy.*

Note that the two channels that may trigger the selection of a new strategy differ significantly in terms of latency and urgency. The exception mechanism provides a fast propagation channel of failures upwards, indicating that the strategy being currently enacted is relying on assumptions that have just been violated. This means that any guarantees on the success of the current strategy in satisfying its requirements are void and a new strategy is urgently required. The second channel is via de knowledge repository. The monitoring of changes in the knowledge repository is a process that incurs comparatively significant delays as the inference of goal model updates based on logged information may be performed sporadically and consume a significant amount of time. The upside of this second channel is that it may predict problems sufficiently ahead of their occurrence, providing time to select pre-computed strategies that may avoid them.

The selection of a behaviour strategy is constrained by the current configuration of the target system (which determines the events and actions that can be used by the strategy) and the alternative configurations that may be reached by enacting one of the pre-computed re-configuration strategies. Furthermore, the selection is informed by preferences defined in the goal model on which OR-refinement resolution is preferred. *Thus, a new search strategy that can be supported by the current UAV configuration may be selected. Alternatively a strategy that no longer picks samples up to avoid the extra consumption produced by load carrying may be chosen. In the later case, in-situ analysis is required and hence a reconfigured UAV with an infra-red camera in place is required. Selecting such a pre-computed behaviour strategy is subject to the availability of a pre-computed reconfiguration strategy that can reach a configuration with an active infra-red camera module.*

The Behaviour Strategy Manager deploys the selected strategy by performing two operations. Firstly, should the selected strategy require a configuration with characteristics that are currently not provided, it commands the Reconfiguration Strategy Manager to deploy an appropriate reconfiguration strategy

(c.f. Master-Slave relationship). Secondly, the manager hot-swaps the current behaviour strategy being executed in the layer below with the newly selected strategy, setting the initial state of the new strategy consistently with the current state of the Target System. Note that should the new strategy be replacing a strategy that is still valid (i.e. no exception has been raised) then the hot-swap procedure may also exploit information extracted from the current state of the strategy to be swapped out.

Should the Behaviour Strategy Manager fail to select a pre-computed behaviour strategy, the manager requests new strategies from the layer above. This may happen, for example, because none of pre-computed strategies it manages have assumptions that are compatible with the actual observed behaviour of the system (e.g., *energy consumption is far worse than what is assumed by any pre-computed strategy*) or that they all rely on unachievable configurations (e.g. *the joint failure of the gripper component and infra-red camera was a operational scenario not considered in any of the pre-computed strategies*).

Reconfiguration Strategy Manager: This entity works similarly to its behaviour counterpart. It stores and manages multiple reconfiguration strategies and selects them for deployment constrained by the availability of components in the Target System while maintaining consistency with the configuration requirements of the current behaviour strategy. Selection is also informed by preferences specified in the goal model. *Consequently, a precision preference may lead to selecting a reconfiguration strategy that attempts to use a GPS rather than a hybrid positioning component.*

When negotiating with the Behaviour Strategy Manager on a pair of strategies to be deployed, the Reconfiguration Strategy Manager takes the slave role, stating the configuration's requirements that are achievable and then selecting an appropriate reconfiguration strategy based on the selection made by the Behaviour Strategy Manager.

There are three channels that can trigger the selection of a new reconfiguration strategy. Two are similar to those that trigger the Behaviour Strategy Manager: An exception from the Reconfiguration Strategy Enactor and a change in the goal model. *Examples of these are the failure of the GPS component triggering a rapid response by the manager which selects an alternative configuration (using the hybrid positioning component) and deploys an appropriate reconfiguration strategy, or an increased response time of the GPS component leading to the decision of changing the positioning system before it (most likely) fails.* The third channel is the request of a new configuration by the Behaviour Strategy Manager (which in turn may have been triggered via the exception mechanism or a change in the goal model).

It is important to note that deployment of new strategies at the Strategy Management layer may respond not only to problems (or foreseen problems) while enacting the current strategies, but also to capitalise on opportunities afforded by a change in the environment. For instance, should a new component become available, or statistics on its performance improve, this would be reflected in the knowledge repository and an alternative preferred pre-computed strategy may be deployed.

2.5 Strategy Enactor

Responsibility: This layer's main responsibility is to execute behaviour and reconfiguration strategies provided by the layer above. Strategy execution involves monitoring the target system and invoking operations on it at appropriate times as defined by the strategy. The layer must also ensure that if the target system reaches a state unexpected by the strategy, this should be reported to the layer above.

Rationale: The aim is to provide a MAPE loop with low analysis latency to allow rapid response to changes in the state of the target system based on pre-computed strategies. In other words to achieve fast adaptation to anticipated behaviour of the target system. IN addition, to allow independent handling of failed assumptions made by either the behaviour or reconfiguration strategies, thereby adapting one strategy in a way that is transparent to the other.

Structure and Behaviour: The layer has two strategy enactors, one for behaviour strategies and the other for reconfiguration strategies. Both enactors work very similarly. They monitor the target system and react to changes in the system by invoking commands on the target system. The decision of which command to execute requires no significant computation. The two enactors do, however, differ in the instrumentation infrastructure they use to monitor and effect the target.

Reconfiguration Strategy Enactor: This entity invokes reconfiguration commands and accesses individual software component status information through an API provided by the Target System layer. The aspects monitored and effected by this enactor tend to be application domain independent; commands and status data are typically related to the component deployment infrastructure and allow operations such as adding, removing and binding components, and checking if they are idle, active, and so on. Commands can include setting operational parameters of components (eg., thread pool).

In addition to sequencing reconfiguration commands, the enactor has to resolve the challenge of ensuring that state information is not lost when the configuration is modified. This can involve ensuring stable conditions such as tranquility [20] or quiescence [19] before change. For instance, *the infra-red camera component can be safely removed from a system if it is isolated (no bindings to or from) and passive (e.g., not processing an image)*.

Behaviour Strategy Enactor: The entity monitors and affects the target system through application domain services provided by the components of the target system. These are accessed via behaviour commands and event abstractions exhibited by the Target System layer. The enactor starts executing the behaviour strategy assuming that there is a configuration in place that can provide the events and commands it requires. *Thus, a new search and analyse behaviour strategy using the gripper is assuming the gripper component is configured.*

Should the behaviour strategy require a different configuration at any point, it must request the configuration change explicitly. In this case a reconfigure

command will be part of the behaviour strategy and the behaviour enactor will command the execution of the reconfiguration strategy stored by the reconfiguration strategy enactor (*e.g., the behaviour strategy folds the arm holding the broken gripper and then requests reconfiguration to incorporate the infra-red camera to only then proceed with in-situ analysis*). Note that in this case the behaviour enactor assumes that the reconfiguration strategy loaded in the other enactor will attempt to reach a target configuration that is consistent with the behaviour strategy.

Assumptions regarding the current configuration and the target configuration of the strategy loaded on the reconfiguration strategy enactor are ensured by the layer above that feeds consistent behaviour and reconfiguration strategies to this layer.

2.6 An Integrating Scenario

In Fig. 2 we informally depict a scenario in which the various components of the MORPH reference architecture interact to resolve unexpected runtime behaviour of our running example, the UAV system.

Consider the UAV is running its mission, being guided along it according to the strategy being enacted by the **Behaviour Strategy Enactor** (see top left of Fig. 2). At some point during the mission, the inference component performs a routine check of the system log and detects that energy usage assumptions as described in the **Goal Model** are unlikely to be valid. As a consequence it updates the **Goal Model** with new assumptions on energy consumption. The change in the **Goal Model** triggers the **Goal Model Manager** to look for new combinations of assumptions and goals that are achievable with the new knowledge by submitting combinations of reconfiguration and behaviour problems to the **Reconfiguration Problem Solver** and the **Behaviour Problem Solver**. For instance, it may fail to find a strategy for achieving the current mission goal (find and retrieve samples for analysis at base location) with the new weaker assumptions on energy consumption because the amount of flight involved would inevitably drain the battery before mission completion. However, it may find that switching from an analyse in-situ strategy is viable but requires a different UAV configuration in which on-board sensors are utilised. Recall (as described in Sect. 1) that for this the robotic arm must be folded to avoid camera obstruction. Furthermore, folding requires landing the UAV before unbinding the arm component and binding hitherto unused sensors. Thus, the **Goal Model Manager** sends a reconfiguration strategy and a behaviour strategy to the **Reconfiguration Strategy Manager** and the **Behaviour Strategy Manager**. These strategies are linked in the sense that the behaviour strategy assumes a particular UAV configuration which can be achieved by the reconfiguration strategy.

While the process of inferring new assumptions and solving new behaviour and reconfiguration problems is happening, the target system (the UAV) continues on its mission, being guided by the **Behaviour Strategy Enactor**. It could be the case that the mission ends successfully despite the prediction made by the inference procedure, that battery consumption was above what was originally assumed. However, in Fig. 2 we assume that at some point a *lowBattery* event

occurs which is not expected by the strategy being enacted by the Behaviour Strategy Enactor (i.e., the strategy assumed that lowBattery could not happen at that point based on the assumptions that were available at the time the strategy was computed). The unexpected event, triggers an exception signalling the Behaviour Strategy Manager that a new strategy is required. The Behaviour Strategy Manager chooses the analyse in-situ strategy which requires a different configuration and negotiates with the Reconfiguration Strategy Manager the selection of an appropriate reconfiguration strategy that may provide this configuration. Having concluding the negotiation, the managers push down the strategies to the Enactors. The Behaviour Strategy Enactor lands the UAV in an appropriate landing spot, folds the arm and then sends a reconfigure command to the Reconfiguration Strategy Enactor. The latter then instantiates software components for controlling the camera and other sensors, binds them to the software architecture and unbinds the arm control component. Once the reconfiguration is finished, the Behaviour Strategy Enactor is notified and the UAV proceeds following the analyse in-situ mission, covering the surveillance area that had not been covered by the previous find and retrieve strategy.

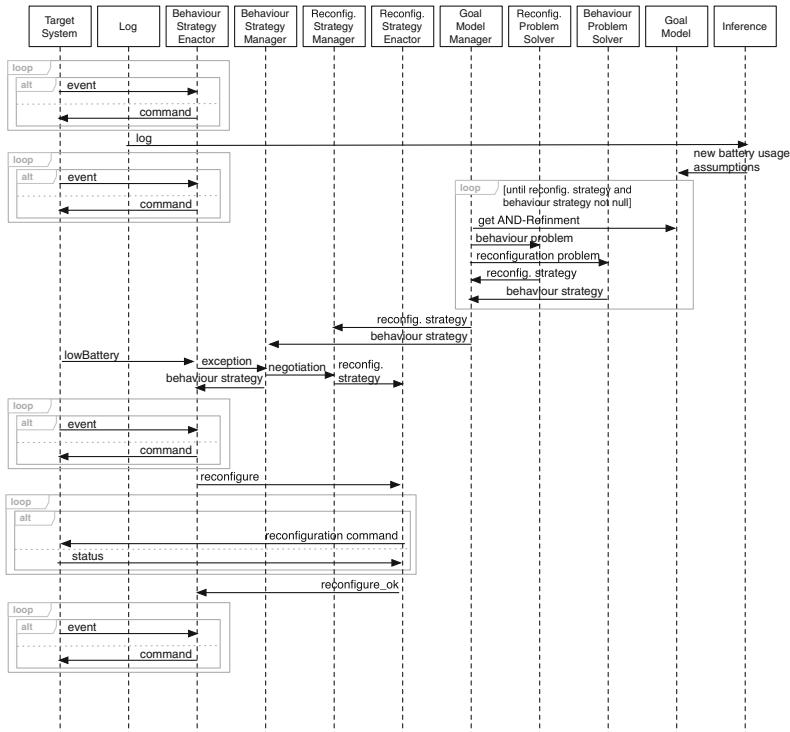


Fig. 2. Example of a MORPH adaptation scenario

3 Relation to Existing Systems and Architectures

3.1 RAINBOW [3]

Rainbow instantiates and refines the MAPE-K architecture providing an extensible framework for sensors and actuators at the interface between the control infrastructure and the target system (see Fig. 3). The architecture recognises the complexity of the interface between the MAPE-K infrastructure (referred to by the authors as the architecture layer) and the component system to be adapted (referred to as the system layer). The Rainbow framework introduces additional infrastructure into the architecture and system layers in addition to accounting for an extra layer between the two: the translation layer. Monitoring is split amongst the three layers: probes are introduced as system layer infrastructure to support observation and measurement of low-level system states. Gauges are part of the architectural infrastructure layer and aggregate information from the probes to update appropriate properties of the knowledge base used for the MAPE activities. The translation layer resolves the abstraction gap between the system layer and the architectural layer, for instance relating abstract component identifiers in the later concrete process and machine identifiers in the former.

Rainbow focuses on achieving self-adaptation through configuration adaptation. Thus, focus is on changing component instances and bindings and also effecting behaviour by changing operational parameters (thread pool size, number of servers, etc.). Indeed, the framework does not account explicitly for automated construction of strategies that control the functional behaviour of the system layer components. Thus, the distinction and coordination between configuration and behaviour control is not elaborated explicitly in the architecture.

MORPH takes inspiration from Rainbow when structuring the instrumentation of the managed system, thus including a translation layer, effector and probes. It also recognizes the complexity of the data accumulation and analysis aspects which in Rainbow are set in an architectural element named Model Manager, while we place it in the Knowledge Repository. The latter serves as the “K” element for multiple MAPE-K loops in MORPH, while in Rainbow the Model Manager services the only MAPE-K loop under execution.

Rainbow’s adaptation engine requires a set of precomputed strategies and tactics, and in this sense, it can be said that the MAPE-K loop it implements falls within the **Strategy Management layer**. Computationally complex construction of strategies is not considered in Rainbow (strategies are assumed to be provided), hence no counterpart to MORPH’s Goal Management layer exists, failing to provide runtime synthesis and structured support for meta-adaptation. Indeed, according to the MORPHreference architecture, the input to Raibow’s Adaptation Engine (the strategies) would be provided by the Goal Management Layer (see Fig. 3).

3.2 PLASMA [11]

PLASMA is a three layered architecture (see Fig. 4) supporting model based adaptation using planning as the core technology for producing adaptation

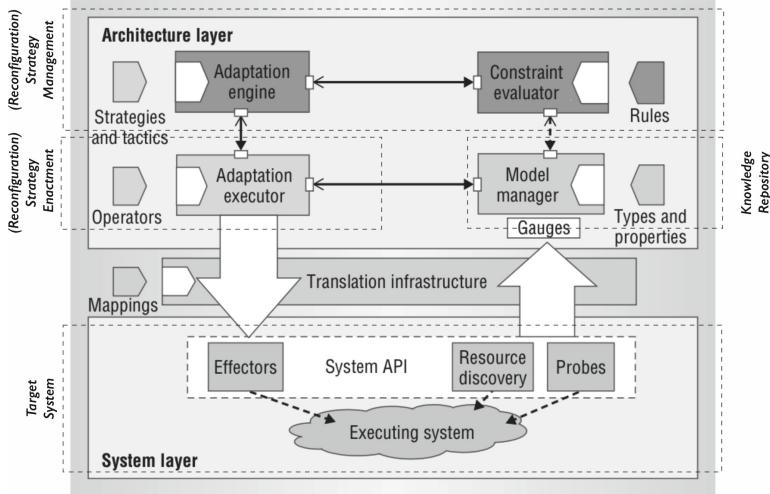


Fig. 3. The Rainbow framework.

strategies. The architecture supports both reconfiguration and behaviour adaptation. The former is achieved through *adaptation plans* while the latter through *application plans*.

In the top PLASMA layer, the panning layer, sit two planners, the Application and the Adaptation Planners which correspond to the behaviour and reconfiguration problem solvers of MORPH. However, in the top PLASMA layer there is no Goal Model Manager that decomposes the system goals into two. Rather, it is assumed that the system goal is first addressed as a behaviour problem and then an adequate reconfiguration is produced for the behaviour strategy that is computed. In a sense, a simplified Goal Model Manager is hard-wired into the layer. This is a key difference with the approach presented herein. In MORPH, the achievable configurations may determine the system goal that can be satisfied. Furthermore, to decouple computationally intensive problem solving from the rest of the system adaptation and execution mechanisms, MORPH allows computing multiple behaviour and reconfiguration strategies, maintaining a many to many relationship between them. This allows having multiple pre-computed adaptation strategy pairs that can be deployed immediately when required.

The middle PLASMA layer is the adaptation layer. Its core element is the Adaptation Analyzer which executes reconfiguration strategies produced by the level above. Thus, the Adaptation Analyzer fits well with MORPH's Reconfiguration Strategy Enactor.

The bottom PLASMA layer is the application layer which contains elements that correspond to what here we call the Target System (the application's component architecture, effectors and probes) and also contains the Application Executor. The Executor enacts behaviour strategies, consequently mapping to MORPH's Behavior Strategy Enactor.

The two bottom PLASMA layers determine a dependency that is not present in MORPH. In PLASMA, it is the reconfiguration adaptation that monitors and commands the behaviour adaptation. We place the behaviour and reconfiguration on equal grounds in all layers and in particular in the Strategy Enactment layer were both enactors are. The dependency introduced in PLASMA entails that the reconfiguration strategy has to put in place the behaviour strategy, forcing a reconfiguration every time a new behaviour strategy is computed.

As discussed previously, to support adaptation independent yet coordinated behaviour and reconfiguration is required. In PLASMA, coordination is achieved by the hard-wired dependency between plan computation and the hierarchical precedence of reconfiguration over behaviour plan enactment. The cost of this coordination is the lack of independent configuration and behaviour adaptation. In MORPH, treating reconfiguration and behaviour planning and enactment as peers supports their independence, coordination is achieved by introducing a Goal Model Manager in the top layer, negotiation in the middle layer and only in the bottom layer, once strategies are guaranteed to be consistent, a temporal coordination dictated by the behaviour strategy.

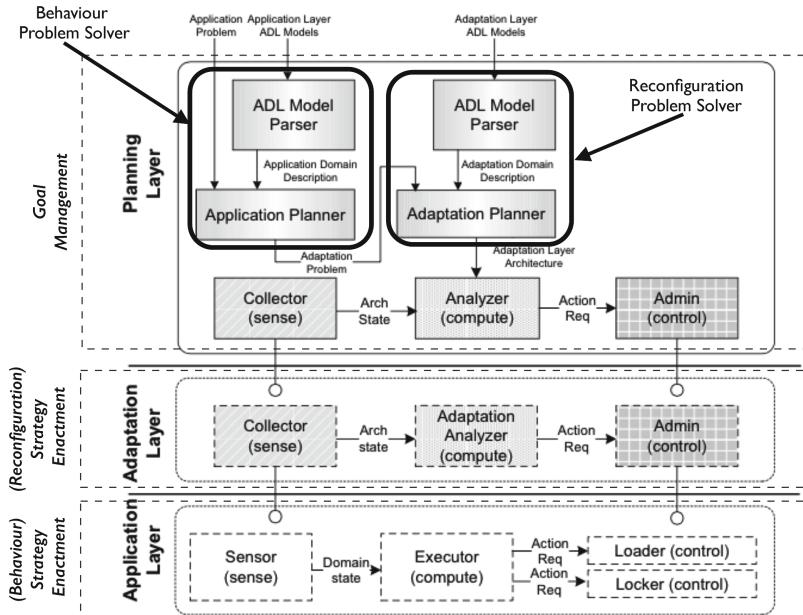


Fig. 4. The PLASMA architecture.

3.3 Three Layer Conceptual Model [13]

The need to deal with hierarchies of control loops in autonomous systems is widely recognised (e.g. [49]). Lower levels are typically low latency loops that

focus on more tactic and stateless objectives that involve less monitored and controlled elements while higher levels tend to focus on more stateful and strategic objectives involving multiple controlled and monitored aspects that require higher latency loops.

The need for hierarchy in architectural self-adaptation is discussed in [13]. A three layer architecture is proposed to provide a separation of concerns and to address a key architectural concern related to dealing with the complexity of run-time construction of adaptation strategies. The architecture structures hierarchically the MAPE-K loops introducing a separation of concerns in which complex, strategic, resource consuming analysis is performed in the top layer (the Goal Management Layer), these plans are managed by the Change Management layer and enacted in the Component Layer.

Although the architecture is conceptual in nature, it does prescribe the kind of control that is effected on the adaptive system by establishing a clear interface between the adaptation infrastructure and the component based system to be adapted. The architecture assumes an interface on which it can take action on the current system configuration by creating and deleting components, binding and unbinding components through their required and provided ports and setting component modes (i.e., configuration parameters). Such interface is used for reconfiguration adaptation. Various instances of this architectural model have been implemented.

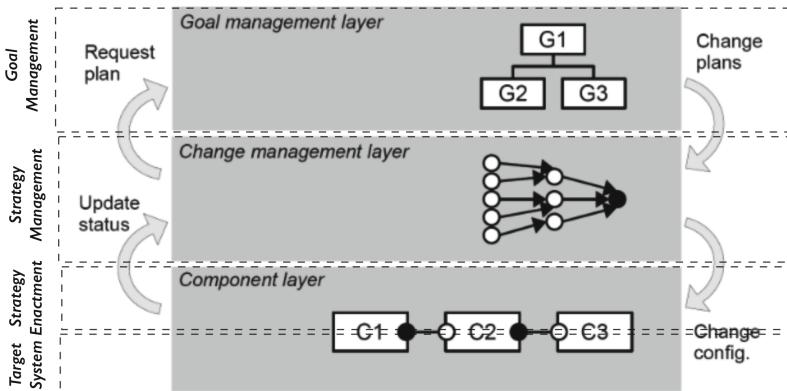


Fig. 5. The three-layer conceptual model [13]

In [43], the three layer reference model is used to also adapt system behaviour. In this case planning is used to produce behaviour strategies in the Goal Management layer. The planner works on automatically inferred behaviour models of the environment. Each plan defines an interface which then must be matched with an appropriate configuration that can provide such interface. Thus a hierarchical relation, as in PLASMA (see above) is established between reconfiguration and behaviour adaptation, which as discussed previously hinders independent and coordinated behaviour and reconfiguration adaptation.

The MORPH reference architecture builds upon this three layered model emphasising the need to make behaviour and reconfiguration control first-class architectural entities and structuring how each works independently and in coordination. Thus, it provides a more refined view of the layers (as depicted in Fig. 5).

4 Prior Experience

The MORPH reference architecture systematically articulates our previous experience in concrete architectures and techniques for self-adaptation. This covers to different degrees, all elements in the reference architecture proposed herein (see Fig. 6).

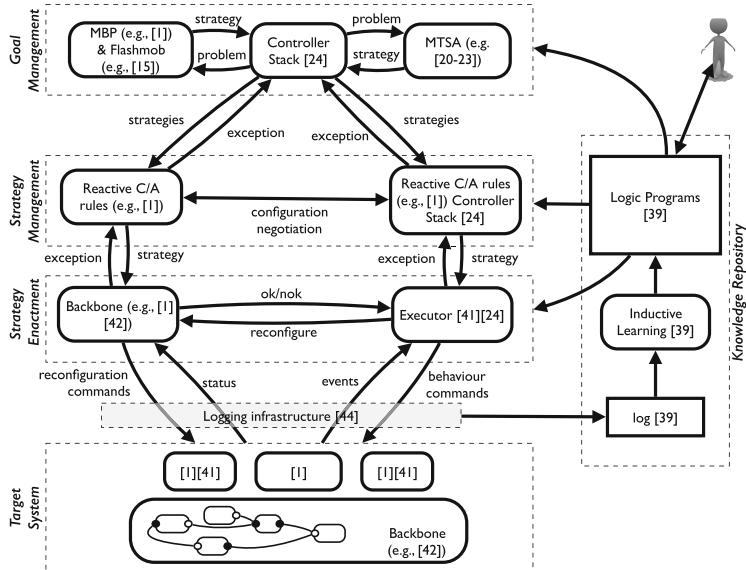


Fig. 6. Prior experience.

We describe the implementation of the Strategy Enactment Layer and how we instrument the Target System to support the Behaviour Strategy Enactor and the Reconfiguration Strategies Enactor in [50] and [1] respectively. In both cases, an interpreter is used to walk through a strategy executing command proxies that are bound to specific application components.

For the enactment of behaviour strategies [50], as the components are normally provided by third parties such as the robot arm manufacturer, each behaviour command may map to an ad-hoc combination of low-level method calls. The adaptive system designer must provide implementations of each high-level command (or event) that will be used in the behaviour strategy.

This transformation of high-level behaviour commands into low-level method calls lives in the translation infrastructure that lies between the Target System and Strategy Enactment layer (not depicted in the MORPH architecture diagram) just as in [3] (see Fig. 3).

The enactment of reconfiguration commands is done in [1] via the Backbone language [34], which is UML 2.0 compatible and resembles Darwin [39]. Thus high-level reconfiguration commands in the reconfiguration strategy are sent via RMI to the Backbone interpreter sitting in the target system (running on a Java VM) which then affects the component architecture appropriately. More details on Backbone can be found in [33].

In [1], we show how the Reconfiguration and Behaviour Strategy Managers can be equipped with general pre-computed strategies that can provide additional robustness to strategies running on a component based systems in which unexpected failures occur. The use of reactive condition-action rules allows re-sensing at regular intervals what the state of the system is to then enact an appropriate rule even if the effect of previously enacted rules was not as expected.

In [29], a more sophisticated management of alternative strategies is used. The Behaviour Strategy Manager stores a hierarchy of strategies, each of which is guaranteed to achieve different goals under different environment assumptions. The hierarchy is organised in terms of the strength of the assumptions each strategy makes (or the risk that each strategy takes). Higher level strategies make bold assumptions on the environment in order to achieve stronger goals. When the Behaviour Strategy Enactor detects that one of an assumption is violated, it raises an exception to the Behaviour Strategy Manager which then puts in place a behaviour strategy further down in the hierarchy. The new strategy will be based on weaker assumptions compatible with the environment behaviour exhibited up to that point, but at the cost of achieving weaker goals. Indeed, the hierarchy can be used to adapt the functional behaviour of the system through graceful degradation when the assumptions of a higher level model are broken, and through progressive enhancement when those assumptions are satisfied or restored.

In [29] we also present formal results showing that pre-computed behaviour strategies must define a simulation order to guarantee graceful degradation and provide seamless progressive enhancement. We also show how a Goal Model Manager can produce a hierarchy of control problems that will result in hierarchy of simulating strategies using standard controller synthesis algorithms implemented in a Behaviour Problem Solver.

The techniques we present in [22] to compute reconfiguration strategies that must address the construction of a distributed configuration in a decentralised fashion fits into the responsibilities of the Reconfiguration Strategy Solver. In [22] we then show how decentralised self-assembly can be implemented over a gossip protocol.

We have worked extensively on synthesis algorithms for the Behaviour Strategy Solver. These involve constructing strategies with formal winning guarantees against adversarial environments [12, 26], dealing through qualitative reasoning with probabilistic failures in the environment [27] and also with partial goal models [28].

In [43] we present how to update a goal model represented as a logic program in the common knowledge repository by using a probabilistic rule learning approach using feedback from the running system in the form of execution traces. Non-monotonic rule learning based on inductive logic programming finds general rules which explain observations in terms of the conditions under which they occur. The updated models are then used to generate new strategies with a greater chance of success.

5 Related Work

The last decade has seen a significant build up of the body of work related to engineering self-adaptive systems [51, 52]. Our work builds on this knowledge, emphasising the need to make behaviour and reconfiguration control first-class architectural entities.

The MAPE-K model [2] shows how to structure a control loop in adaptive systems. The four key activities (Monitor, Analysis, Plan and Execute) are performed over a shared data structure that captures the knowledge required for adaptation. The MAPE-K model does not prescribe what knowledge is to be captured nor what aspect of the system is to be controlled. Thus, there is no explicit treatment or distinction between configuration and behaviour adaptation let alone prescribed mechanisms for dealing with coordinated and independent configuration and behaviour adaptation. We design each layer of the reference architecture as a MAPE-K control loop, resulting in a hierarchical control loop structure as in [53].

As discussed in Sect. 2, the architecture proposed builds on those of [2, 3, 13] and others (e.g., [14–16]). However, to the best of our knowledge existing work does not provide support for both independent and also coordinated structural and behavioural adaptation at the architectural level with exception of [54] and [11] where behaviour and dependency injection strategies are computed separately but are forced to be executed serially.

The MORPH reference architecture is geared towards the use of strategies derived from the field of control engineering referred to as discrete event dynamic system (DEDS) control [55] which naturally fits over the system abstractions used at the architecture level, which is the level we envisage self-adaptation supported by MORPH to operate. DEDS are discrete-state, event-driven systems of which the state evolution depends entirely on the occurrence of discrete events over time. The field builds on, amongst others, supervisory control theory [8], queueing theory [56], and reactive planning [57].

Automated construction of DEDS control strategies have been applied for self-adaptation in many different forms. For instance, in [10] temporal planning is used to produce reconfiguration strategies that do not consider structural constraints and the status of components when applying reconfiguration actions. In [11], an architecture description language (ADL) and a planning-as-model-checking are used to compute and enact reconfiguration strategies. In [9], quantitative analysis and planning are used to compute evolution strategies. In [23–25]

automatic generation of event-based coordination strategies is applied for run-time adaptation of deadlock-free mediators. In [58], a learning technique (the L* algorithm [59]) is applied for automatically generating components behaviour. Note that strategies do not have to be necessarily temporal sequencing of actions or commands. For instance, in [60] reconfiguration strategies used are one-step component parameter changes.

The use of techniques based on control theory for continuous-variable dynamic systems (CVDS) has also significant application to self-adaptation and is also used at the architectural level [5, 18, 61–63]. Existing techniques of continuous control theory applied to adaptation are single-input single-output (SISO) or multiple-input single-output (MIMO) at best. This differs from discrete event control which tends to be multiple-input multiple-output (MIMO). The controlled variable for these approaches is typically related to an operational parameter of the system configuration (e.g. Thread pool size [64], processor clock speed [4], self-imposed thread sleeps [62], accepted requests per time [65]) thus falling into the category of reconfiguration strategies in our reference architecture. A noteworthy example of such approach is [4], which uses a three tiered scheme that has some parallels with one of the design concerns addressed in this paper. The work in [4] can be understood as a three layered continuous control framework in which the **Strategy Enactor** implements a linear continuous control strategy. The **Strategy Management Layer** manages a family of control strategies in which the value of a constant used in the bottom layer is tweaked when the system diverges beyond a threshold, and the **Goal Management Layer** propagates downwards a family of control strategies computed from scratch based on a higher latency **Inference** process that analyses the execution log.

Goal modelling notations have been identified as central to self-adaptation [66]. We envision using techniques for modelling adaptation requirements (e.g., [67–69]) for reasoning about adaptation in the Knowledge Repository. In addition, having a flexible representation of the requirements and the run-time behaviour of the system is also desired. We envisage using approaches such as [70] where an executable modelling language for runtime execution of models (EUREMA) facilitates seamless adaptation.

The need for coordinated control loops to deal with complex self-adaptation scenarios has been identified in [2, 71] amongst others. The MORPH architecture has a hierarchy of MAPE-K loops provided by the **Goal Manager**, **Strategy Manager** and **Strategy Enactor** layers. Furthermore, the bottom two layers are actually implementing two concurrent, yet coordinated, control loops: one for behaviour and the other for reconfiguration control.

The problem of strategy update, required when a strategy in the enactment layer is replaced by a new one, is a crucial part of runtime adaptation and has been studied extensively both in the context of reconfiguration strategies (e.g., [19, 20, 72]) and behaviour strategies [21].

6 Conclusions

An architectural approach to self-adaptive systems involves runtime change to both the system configuration and behaviour. In this paper we propose MORPH, a three-layered reference architecture which separates these aspects, supporting independent and coordinated reconfiguration and behaviour adaptation. This proposal builds on the extensive research work conducted by ourselves and others which provides various techniques and software entities which plug into the architecture; as such MORPH helps to identify and compare the architectural entities investigated and implemented in different systems. There is as yet no comprehensive system covering the full scope of MORPH. Although this is not essential, as demonstrated by the achievements of existing systems, we plan to investigate this further to provide a wider range of reconfiguration and behaviour strategies.

References

1. Sykes, D., Heaven, W., Magee, J., Kramer, J.: From goals to components: a combined approach to self-management. In: Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems, SEAMS 2008, pp. 1–8. ACM, New York (2008)
2. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003). <https://doi.org/10.1109/MC.2003.1160055>
3. Garlan, D., Cheng, S., Huang, A., Schmerl, B.R., Steenkiste, P.: Rainbow: architecture-based self-adaptation with reusable infrastructure. IEEE Comput. **37**(10), 46–54 (2004)
4. Filieri, A., Hoffmann, H., Maggio, M.: Automated design of self-adaptive software with control-theoretical formal guarantees. In: Jalote, P., Briand, L.C., van der Hoek, A. (eds.) 36th International Conference on Software Engineering, ICSE 2014, Hyderabad, India, 31 May–07 June 2014, ACM, 2014, pp. 299–310. <https://doi.org/10.1145/2568225.2568272>
5. Parekh, S.: Feedback control techniques for performance management of computing systems. Ph.D. dissertation, Seattle, WA, USA (2010)
6. Baresi, L., Guinea, S.: Dynamo and self-healing BPEL compositions. In: Companion to the Proceedings of the 29th International Conference on Software Engineering, ICSE COMPANION 2007, pp. 69–70. IEEE Computer Society, Washington, DC. <https://doi.org/10.1109/ICSECOMPANION.2007.31>
7. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc., San Francisco (2004)
8. Ramadge, P.J., Wonham, W.M.: The control of discrete event systems. Proc. IEEE **77**(1), 81–98 (1989)
9. Kang, S., Garlan, D.: Architecture-based planning of software evolution. Int. J. Softw. Eng. Knowl. Eng. **24**(2), 211–242 (2014)
10. Arshad, N., Heimbigner, D., Wolf, A.L.: Deployment and dynamic reconfiguration planning for distributed software systems. Softw. Qual. J. **15**(3), 265–281 (2007)

11. Tajalli, H., Garcia, J., Edwards, G., Medvidovic, N.: Plasma: a plan-based layered architecture for software model-driven adaptation. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE 2010, pp. 467–476. ACM, New York (2010). <https://doi.org/10.1145/1858996.1859092>
12. D’Ippolito, N., Braberman, V.A., Piterman, N., Uchitel, S.: Synthesizing nonanomalous event-based controllers for liveness goals. *ACM Trans. Softw. Eng. Methodol.* **22**(1), 9 (2013)
13. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, 23–25 May 2007, Minneapolis, MN, USA, pp. 259–268 (2007)
14. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: Towards architecture-based self-healing systems. In: Proceedings of the First Workshop on Self-healing Systems, WOSS 2002, pp. 21–26. ACM, New York (2002)
15. Batista, T., Joolia, A., Coulson, G.: Managing dynamic reconfiguration in component-based systems. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, pp. 1–17. Springer, Heidelberg (2005). https://doi.org/10.1007/11494713_1
16. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. *IEEE Intell. Syst.* **14**(3), 54–62 (1999)
17. Inverardi, P., Tivoli, M.: Software architecture for correct components assembly. In: Bernardo, M., Inverardi, P. (eds.) SFM 2003. LNCS, vol. 2804, pp. 92–121. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39800-4_6
18. Leva, A., Maggio, M., Papadopoulos, A.V., Terraneo, F.: Control-Based Operating System Design, Ser. Control Engineering Series, IET (2013)
19. Kramer, J., Magee, J.: The evolving philosophers problem: dynamic change management. *IEEE Trans. Softw. Eng.* **16**(11), 1293–1306 (1990). <https://doi.org/10.1109/32.60317>
20. Vandewoude, Y., Ebraert, P., Berbers, Y., D’Hondt, T.: Tranquility: a low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. Softw. Eng.* **33**(12), 856–868 (2007). <https://doi.org/10.1109/TSE.2007.70733>
21. Ghezzi, C., Greenyer, J., La Manna, V.P.: Synthesizing dynamically updating controllers from changes in scenario-based specifications. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2012, pp. 145–154. IEEE Press, Piscataway (2012). <http://dl.acm.org/citation.cfm?id=2666795.2666819>
22. Sykes, D., Magee, J., Kramer, J.: FlashMob: distributed adaptive self-assembly. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, pp. 100–109. ACM, New York (2011)
23. Di Marco, A., Inverardi, P., Spalazzese, R.: Synthesizing self-adaptive connectors meeting functional and performance concerns. In: Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2013, pp. 133–142. IEEE Press, Piscataway (2013)
24. Bennaceur, A., Inverardi, P., Issarny, V., Spalazzese, R.: Automated synthesis of connectors to support software evolution. *ERCIM News* **2012**, 88 (2012)

25. Issarny, V., Bennaceur, A., Bromberg, Y.-D.: Middleware-layer connector synthesis: beyond state of the art in middleware interoperability. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 217–255. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21455-4_7
26. D’Ippolito, N., Fischbein, D., Chechik, M., Uchitel, S.: MTSA: the modal transition system analyser. In: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15–19 September 2008, L’Aquila, Italy, pp. 475–476. IEEE (2008). <https://doi.org/10.1109/ASE.2008.78>
27. D’Ippolito, N., Braberman, V.A., Piterman, N., Uchitel, S.: Synthesis of live behaviour models for fallible domains. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, 21–28 May 2011, pp. 211–220 (2011)
28. D’Ippolito, N., Braberman, V.A., Piterman, N., Uchitel, S.: The modal transition system control problem. In: FM 2012: Formal Methods - Proceedings of 18th International Symposium, Paris, France, 27–31 August 2012, pp. 155–170 (2012)
29. D’Ippolito, N., Braberman, V.A., Kramer, J., Magee, J., Sykes, D., Uchitel, S.: Hope for the best, prepare for the worst: multi-tier control for adaptive systems. In: Jalote, P., Briand, L.C., van der Hoek, A. (eds.) 36th International Conference on Software Engineering, ICSE 2014, Hyderabad, India, 31 May–07 June 2014, pp. 688–699. ACM (2014)
30. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. *Commun. ACM* **15**(12), 1053–1058 (1972). <https://doi.org/10.1145/361598.361623>
31. Braberman, V.A., D’Ippolito, N., Kramer, J., Sykes, D., Uchitel, S.: MORPH: a reference architecture for configuration and behaviour self-adaptation. In: Filieri, A., Maggio, M. (eds.) Proceedings of the 1st International Workshop on Control Theory for Software Engineering, CTSE@SIGSOFT FSE 2015, Bergamo, Italy, 31 August–04 September 2015, pp. 9–16. ACM (2015). <https://doi.org/10.1145/2804337.2804339>
32. Hirsch, D., Kramer, J., Magee, J., Uchitel, S.: Modes for software architectures. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344, pp. 113–126. Springer, Heidelberg (2006). https://doi.org/10.1007/11966104_9
33. McVeigh, A., Kramer, J., Magee, J.: Evolve: tool support for architecture evolution. In: Taylor, R.N., Gall, H., Medvidovic, N. (eds.) Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, 21–28 May 2011, pp. 1040–1042. ACM (2011). <https://doi.org/10.1145/1985793.1985990>
34. McVeigh, A., Kramer, J., Magee, J.: Using resemblance to support component reuse and evolution, pp. 49–56 (2006). <https://doi.org/10.1145/1181195.1181206>
35. Lamsweerde, A.V.: Goal-oriented requirements engineering: a guided tour. In: Proceedings of the Fifth IEEE International Symposium on Requirements Engineering, IEEE Computer Society Washington, DC, USA, p. 0249. IEEE Computer Society, Los Alamitos (2001)
36. Yu, E.S.K.: Towards modeling and reasoning support for early-phase requirements engineering. In: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering, RE 1997, pp. 226–235. IEEE Computer Society, Washington, DC (1997). <http://dl.acm.org/citation.cfm?id=827255.827807>
37. Jackson, M.: The world and the machine. In: Proceedings of the 17th International Conference on Software Engineering, ICSE 1995, pp. 283–292. ACM, New York (1995). <https://doi.org/10.1145/225014.225041>

38. Jackson, M.: Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices. ACM Press/Addison-Wesley Publishing Co., New York (1995)
39. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying distributed software architectures. In: Schäfer, W., Botella, P. (eds.) ESEC 1995. LNCS, vol. 989, pp. 137–153. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60406-5_12
40. Garlan, D., Monroe, R., Wile, D.: ACME: an architecture description interchange language. In: Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON 1997, p. 7. IBM Press (1997). <http://dl.acm.org/citation.cfm?id=782010.782017>
41. Heitmeyer, C.L.: Software Cost Reduction. John Wiley & Sons Inc. (2002). <https://doi.org/10.1002/0471028959.sof307>
42. Harel, D.: Statecharts: a visual formalism for complex systems. *Sci. Comput. Program.* **8**(3), 231–274 (1987). <https://doi.org/10.1016/0167-6423.90035>–90039
43. Sykes, D., Corapi, D., Magee, J., Kramer, J., Russo, A., Inoue, K.: Learning revised models for planning in adaptive systems. In: Proceedings of the 2013 International Conference on Software Engineering, Ser., ICSE 2013, pp. 63–71. IEEE Press, Piscataway (2013)
44. Hollnagel, E.: Human Reliability Analysis: Context and Control, A Volume in the Computers and People Series. Academic Press (1993). <https://books.google.com.ar/books?id=jGtRAAAAMAAJ>
45. de Lemos, R., Fields, B., Saeed, A.: Analysis of safety requirements in the context of system faults and human errors. In: Proceedings of the 1995 International Symposium and Workshop on Systems Engineering of Computer Based Systems, pp. 374–381 (1995)
46. Bertoli, P., Cimatti, A., Pistore, M., Roveri, M., Traverso, P.: MBP: a model based planner. In: Proceedings of the IJCAI 2001 Workshop on Planning under Uncertainty and Incomplete Information (2001)
47. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NUSMV: a new symbolic model checker. In: STTT, vol. 2, no. 4, pp. 410–425 (2000). <https://doi.org/10.1007/s100090050046>
48. Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., Traverso, P.: Planning and monitoring web service composition. In: Bussler, C., Fensel, D. (eds.) AIMS 2004. LNCS (LNAI), vol. 3192, pp. 106–115. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30106-6_11
49. Gat, E., Bonnasso, R.P., Murphy, R.: On three-layer architectures. In: Artificial Intelligence and Mobile Robots, pp. 195–210. AAAI Press (1997)
50. Braberman, V.A., D’Ippolito, N., Piterman, N., Sykes, D., Uchitel, S.: Controller synthesis: from modelling to enactment. In: 35th International Conference on Software Engineering, ICSE 2013, San Francisco, CA, USA, 18–26 May 2013, pp. 1347–1350 (2013)
51. Eighth IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2014, London, United Kingdom, 8–12 September 2014. IEEE (2014). <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6998166>
52. Engels, G., Bencomo, N. (eds.): Proceedings of 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, Hyderabad, India, 2–3 June 2014. ACM (2014). <http://dl.acm.org/citation.cfm?id=2593929>
53. Poole, D.L., Mackworth, A.K.: Artificial Intelligence: Foundations of Computational Agents. Cambridge University Press, New York (2010)

54. da Silva, C.E., de Lemos, R.: A framework for automatic generation of processes for self-adaptive software systems. *Informatica (Slovenia)* **35**(1), 3–13 (2011)
55. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems, 2nd edn. Springer, Heidelberg (2010). <https://doi.org/10.1007/978-0-387-68612-7>
56. Allen, A.O.: Probability, Statistics, and Queueing Theory with Computer Science Applications. Academic Press Inc., Orlando (1978)
57. Cimatti, A., Pistore, M., Roveri, M., Traverso, P.: Weak, strong, and strong cyclic planning via symbolic model checking. *Artif. Intell.* **147**(1–2), 35–84 (2003)
58. Giannakopoulou, D., Păsăreanu, C.S.: Context synthesis. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 191–216. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21455-4_6
59. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_24
60. Swanson, J., Cohen, M.B., Dwyer, M.B., Garvin, B.J., Firestone, J.: Beyond the rainbow: self-adaptive failure avoidance in configurable systems. In: Cheung, S., Orso, A., Storey, M.D. (eds.) Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, 16–22 November 2014, pp. 377–388. ACM (2014). <https://doi.org/10.1145/2635868.2635915>
61. Brun, Y., et al.: Engineering self-adaptive systems through feedback loops. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_3
62. Diao, Y., Hellerstein, J.L., Parekh, S., Griffith, R., Kaiser, G.E., Phung, D.: A control theory foundation for self-managing computing systems. *IEEE J. Sel. Areas Commun.* **23**(12), 2213–2222 (2006)
63. Kephart, J., Chess, D.: The vision of autonomic computing. *Computer* **36**(1), 41–50 (2003)
64. Lu, C., Lu, Y., Abdelzaher, T.F., Stankovic, J.A., Son, S.H.: Feedback control architecture and design methodology for service delay guarantees in web servers. *IEEE Trans. Parallel Distrib. Syst.* **17**(9), 1014–1027 (2006)
65. Kihl, M., Robertsson, A., Andersson, M., Wittenmark, B.: Control-theoretic analysis of admission control mechanisms for web server systems. *World Wide Web* **11**(1), 93–116 (2008)
66. Cheng, B.H.C., et al.: Software engineering for self-adaptive systems: a research roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_1
67. Baresi, L., Pasquale, L.: Live goals for adaptive service compositions. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2010, pp. 114–123. ACM, New York (2010). <https://doi.org/10.1145/1808984.1808997>
68. Nakagawa, H., Ohsuga, A., Honiden, S.: Constructing self-adaptive systems using a KAOS model. In: Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW 2008, pp. 132–137. IEEE Computer Society, Washington, DC (2008). <https://doi.org/10.1109/SASOW.2008.35>

69. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.-M.: Relax: a language to address uncertainty in self-adaptive systems requirement. *Requir. Eng.* **15**(2), 177–196 (2010). <https://doi.org/10.1007/s00766-010-0101-0>
70. Vogel, T., Giese, H.: Model-driven engineering of self-adaptive software with Eurema. *ACM Trans. Auton. Adapt. Syst.* **8**(4), 1801–1833 (2014)
71. Vromant, P., Weyns, D., Malek, S., Andersson, J.: On interacting control loops in self-adaptive systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, pp. 202–207. ACM, New York (2011)
72. Cook, J.E., Dage, J.A.: Highly reliable upgrading of components. In: Proceedings of the 21st International Conference on Software Engineering, ICSE 1999, pp. 203–212. ACM, New York (1999). <https://doi.org/10.1145/302405.302466>
73. Jalote, P., Briand, L.C., van der Hoek, A. (eds.): 36th International Conference on Software Engineering, ICSE 2014, Hyderabad, India, 31 May–07 June 2014. ACM (2014)
74. Bernardo, M., Issarny, V. (eds.): SFM 2011. LNCS, vol. 6659. Springer, Heidelberg (2011). <https://doi.org/10.1007/978-3-642-21455-4>

MOSES: A Platform for Experimenting with QoS-Driven Self-Adaptation Policies for Service Oriented Systems

Valeria Cardellini¹(✉), Emiliano Casalicchio², Vincenzo Grassi¹, Stefano Iannucci¹, Francesco Lo Presti¹, and Raffaela Mirandola³

¹ Dip. di Ingegneria Civile e Ingegneria Informatica,
Università di Roma “Tor Vergata”, 00133 Roma, Italy

{cardellini, iannucci}@ing.uniroma2.it, vincenzo.grassi@uniroma2.it,
lopresti@info.uniroma2.it

² Department of Computer Science and Engineering,
Blekinge Institute of Technology, 37141 Karlskrona, Sweden
emiliano.casalicchio@bth.se

³ Dip. di Elettronica, Informazione e Bioingegneria,
Politecnico di Milano, 20133 Milano, Italy
raffaela.mirandola@polimi.it

Abstract. Architecting software systems according to the service-oriented paradigm, and designing runtime self-adaptable systems are two relevant research areas in today’s software engineering. In this chapter we present MOSES, a software platform supporting QoS-driven adaptation of service-oriented systems. It has been conceived for service-oriented systems architected as composite services that receive requests generated by different classes of users. MOSES integrates within a unified framework different adaptation mechanisms. In this way it achieves a greater flexibility in facing various operating environments and the possibly conflicting QoS requirements of several concurrent users. Besides providing its own self-adaptation functionalities, MOSES lends itself to the experimentation of alternative approaches to QoS-driven adaptation of service-oriented systems thanks to its modular architecture.

1 Introduction

The *service-oriented architecture* (SOA) paradigm encourages the realization of new software systems by composing network-accessible loosely-coupled services. Given their intrinsic characteristics, SOA-based systems are characterized by a continuous evolution: providers may modify the exported services; new services may become available; existing services may be discontinued by their providers; usage profiles may change over time; variations in the communication infrastructure may impact the reachability of existing services [7]. As a consequence,

E. Casalicchio—Work done when the author was affiliated with Università di Roma “Tor Vergata”.

SOA-based systems represent a typical domain where the introduction of self-adaptation features can give significant advantages in fulfilling and maintaining over time a given set of non-functional requirements concerning the delivered quality of service (QoS).

Within this framework, we present in this chapter MOSES (*M*odel-based *S*elf-adaptation of *SOA* systems), a software platform for QoS-driven runtime self-adaptation of service oriented systems. MOSES is tailored for a utilization scenario where a SOA system architected as a composite service needs to fulfill either single requests or a sustained traffic of requests generated in both cases by several classes of users. Within this scenario, MOSES determines the most suitable system configuration for a given operating environment by solving an optimization problem, derived from a model of the composite service and of its environment. The adopted model allows MOSES to integrate in a unified framework both the selection of the set of concrete services to be used in the composition, and (possibly) the selection of the coordination pattern for multiple functionally equivalent services, where the latter allows obtaining QoS levels that could not be achieved by using single services. In this respect, MOSES is a step forward with respect to several proposed approaches for runtime SOA systems adaptation, which limit the range of their actions to the selection of single services to be used in the composition.

MOSES is architected as a centralized broker, which advertises and offers to prospective users the composite service it manages. To fulfill the functional and non-functional requirements agreed on with the service users, MOSES implements the functionalities envisaged in the MAPE-K (Monitor, Analyze, Plan, Execute, and Knowledge) reference model for autonomic systems [20], thus acting as a runtime controller for the composite service. Special care has been given in implementing these functionalities according to a modular architecture, so that different implementations can be easily plugged into the overall MOSES framework. This makes MOSES amenable for the experimentation of different adaptation policies aimed at maintaining the QoS delivered by a composite service. As shown in the following sections, this has allowed us to experiment with adaptation policies tailored for different utilization scenarios (e.g., with more or less stringent QoS requirements, or different models of service demand), and can allow other researchers to experiment with their own policies. Therefore, MOSES can be of help to get confidence about the ability of a given adaptation policy in providing guarantees that the system QoS requirements are satisfied. In this respect, in Sect. 2.3 we briefly discuss MOSES in terms of the benchmarking criteria proposed in Chap. 2 of this book (*Perpetual Assurances in Self-Adaptive Systems*), to assess the MOSES capabilities in providing assurances for SOA systems and in Sect. 5 we compare MOSES with existing approaches according to these criteria.

The main features and the overall design of MOSES have been already presented in [12]. In this chapter, we provide the detailed design of the MOSES platform and present some insights of its implementation, with the goal of providing a framework that can be used and modified by the self-adaptive systems

software engineering community. To this end, we release the source code of MOSES, which is freely available with an opensource license at <http://www.ce.uniroma2.it/moses>. We hope that this software can provide a platform for developing and experimenting with different QoS-driven adaptation mechanisms exploited in the context of service oriented systems. On the web site, along with the source code, we also provide the documentation of the MOSES modules, so that the interested researchers and practitioners can either modify the existing mechanisms or plug their own into MOSES.

The remaining of this chapter is organized as follows. In Sect. 2 we classify MOSES within a frame of reference for QoS-driven self-adaptation of SOA systems and characterize its adaptation capabilities with respect to a case study that has been proposed as a testbed for the benchmarking criteria proposed in this book (see Chap. 2 of this book (*Perpetual Assurances in Self-Adaptive Systems*)). In Sect. 3 we outline the MOSES architecture and the main tasks of its components. In Sect. 4 we dive into the MOSES implementation. In Sect. 5 we review related work, focusing on frameworks similar to MOSES, and discuss their adaptation capabilities according to the benchmarking criteria proposed in this book. Finally, we conclude in Sect. 6, summarizing some lessons learned with the MOSES development and presenting directions for future work.

2 MOSES Framework

2.1 Problem Space Characterization

To better delineate the contribution given by MOSES, we briefly outline a characterization of the problem space of QoS-driven self-adaptation for the SOA domain, taken from [12], providing a frame of reference for MOSES itself and for other contributions in the existing literature. Figure 1 summarizes the main concepts of this characterization, which are briefly described in the following. We refer to [12] for a thorough discussion of these concepts. The taxonomy is built around some basic questions and challenges for the whole domain of self-adaptive software systems [36], and the corresponding possible answers based on the specific features of the SOA domain, with special emphasis on QoS aspects:

- *why* should adaptation be performed (which are its goals);
- *when* should adaptation actions be applied;
- *where* the adaptation should occur (in which part of the system) and *what* elements should be changed;
- *how* should adaptation be implemented (by means of which actions);
- *who* should be involved in the adaptation process.

Why. Non functional requirements concerning the delivered QoS and cost, are usually expressed in the SOA domain by *Service Level Agreements* (SLA) [27]. In a stochastic setting, a SLA can concern guarantees about the *average value* of quality attributes, or about the *higher moments* or *percentiles* of these attributes.

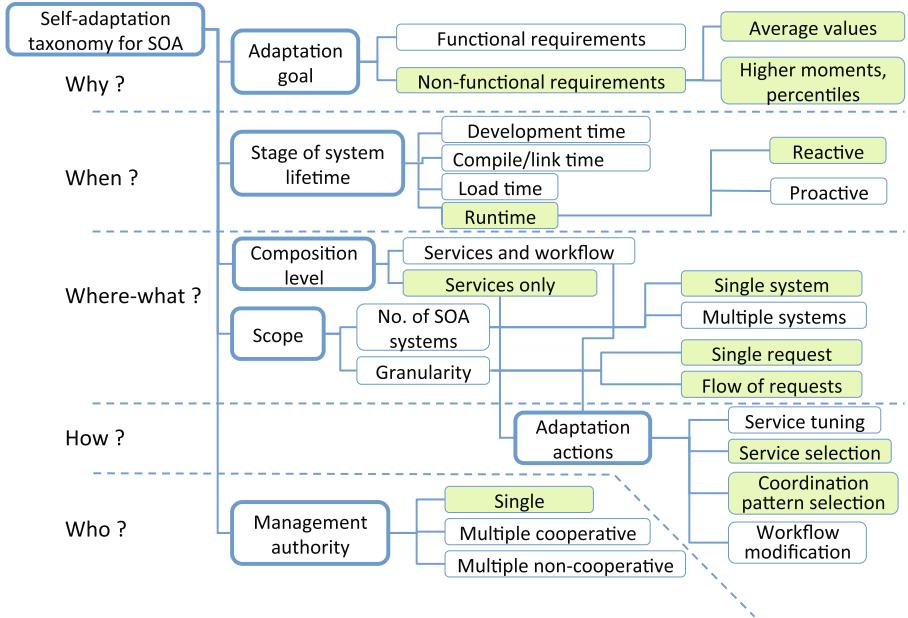


Fig. 1. Taxonomy of self-adaptation for SOA.

When. In the SOA domain, adaptation usually occurs at runtime, and can be *reactive* or *proactive*. In reactive approaches, collected data are evaluated and the adaptation to problematic events takes place after they have occurred. Proactive approaches take advantages of predictive models to detect in advance the need of changes, thus possibly invoking the adaptation before the SLA violation could actually happen.

Where-What. From the viewpoint of *service composition*, adaptation in the SOA domain may take place at two different levels: (i) *concrete services only*: the adaptation only involves the concrete composition, acting on the implementation each abstract task is bound to, leaving unchanged the composition logic (*i.e.*, the overall abstract composition); (ii) *services and workflow*: the adaptation involves both the concrete and abstract composition; in particular, the composition logic can be altered.

From the viewpoint of its *scope*, adaptation in the SOA domain can be characterized in terms of: (i) the *granularity* level at which adaptation is performed; (ii) the *number of SOA systems* operating in the same environment that are directly involved in the adaptation process. In the first case (granularity), adaptation can concern the fulfilment of requirements for a *single service request*, or the overall *flow of requests* addressed to a SOA system. In the second case (number of SOA systems), adaptation can focus on a *single SOA system*, or *multiple systems* competing for overlapping sets of services.

How. Possible adaptation actions include: (i) *service tuning*; (ii) *service selection* (selection for each abstract task of a matching *single* operation offered by a concrete service); (iii) *coordination pattern selection* (selection for each abstract task of a *set* of functionally equivalent operations offered by different concrete services, coordinating them according to some spatial or temporal redundancy pattern, e.g., k -out-of- n); (iv) *workflow modification* (modification or reconfiguration of the workflow composition logic, e.g., through the activation and deactivation of features in the variability model [2]).

Who. In case of multiple SOA systems, their adaptation can be under the control of: (i) a *single authority*; (ii) *multiple cooperating authorities*; (iii) *multiple non cooperating authorities*.

2.2 System Model

MOSES manages the runtime self-adaptation of composite services. The overall utilization scenario where MOSES is intended to operate can be outlined as follows (we refer to [12] for a more detailed description).

Managed System. MOSES manages the class of SOA systems consisting of composite services whose orchestration logic can be described by a structured workflow built using the following composition rules: (i) *sequential* composition; (ii) *conditional selection*; (iii) *conditional iteration*; (iv) *fork-join* parallel composition.

Service Demand. MOSES focuses on a scenario where several classes of users address their requests to a composite service. Each class may have its own QoS requirements, and negotiates a corresponding SLA with the system. The current MOSES implementation includes modules that allow the management of SLAs concerning either each *single* request, or the overall *flows* of requests generated by different users.

Environment Dynamics. The MOSES goal is to keep the managed system able to fulfil the QoS requirements of its users despite the occurrence of variations in the system operating environment. Currently, tracked variations that could trigger an adaptation action include: (i) the arrival/departure of a user with the associated QoS requirements; (ii) observed variations in the QoS attributes of the concrete services used by the service composition; (iii) addition/removal of concrete services implementing some task of the abstract composition; (iv) variations in the usage profile of the tasks in the abstract composition.

Adaptation Actions. MOSES dynamically binds each abstract service specified in the composite service workflow to a concrete implementation taken from a

known set of available services. When the current bindings result no longer adequate to fulfil the existing QoS requirements (as a consequence of an occurred environment variation) the adaptation action performed by MOSES consists in a change of some of those bindings. Given an abstract service S in the workflow, possible changes currently include: (i) binding S to an implementation consisting of a single concrete service; (ii) binding S to an implementation consisting of a set of functionally equivalent services, coordinated according to a sequential *try until success* pattern (greater reliability and higher cost with respect to single service binding); (iii) binding S to an implementation consisting of a set of n functionally equivalent services, coordinated according to a parallel *1-out-of- n* pattern (smaller response time and higher cost with respect to single service binding).

We point out that actually the MOSES adaptation actions include not only the dynamic deterministic binding of an abstract service to a concrete implementation, but also the dynamic probabilistic binding, where the implementation is probabilistically selected from a suitable set of alternatives. This allows, for example, the probabilistic partitioning of requests in a flow among different implementations, thus giving more flexibility in achieving the required QoS level. More details can be found in [12, 13].

QoS Requirements. MOSES offers to the prospective concurrent users of the composite service it manages the possibility of specifying QoS requirements expressed as min/max thresholds on the average value or percentile of some QoS attributes. In the current MOSES implementation, considered QoS attributes include the service *response time*, *cost* and *reliability*.

Besides trying to fulfil the threshold-based QoS requirements of the composite service users, MOSES can also take into consideration an additional requirement concerning the optimization of a function of the composite service QoS attributes. Depending on the utilization scenario, this (utility) function could be aimed at optimizing specific QoS attributes for different classes of users (*e.g.*, minimizing their experienced response time) and/or it could be aimed at optimizing the MOSES own utility, *e.g.*, minimizing the overall cost to offer the composite service (that would maximize the MOSES owner incomes). These different optimization goals could be possibly conflicting, thus leading to a multi-objective optimization problem. MOSES deals with it by using the Simple Additive Weighting (SAW) technique [18].

To fulfil these requirements, MOSES relies on the availability of a set of concrete services that can be used to instantiate the abstract services in the managed composite service. Each concrete service in this set is assumed to offer a SLA concerning the guarantees it gives about its QoS attributes, provided that the load generated by the user does not exceed a given threshold (specified in the same SLA). The guarantees specified in the SLA are analogous to those managed by MOSES, *i.e.*, concerning min/max thresholds on the average value or percentile of the QoS attributes.

To manage the resulting overall set of QoS requirements and constraints, and to determine the suitable adaptation actions, MOSES instantiates and solves a suitable instance of the following optimization problem template:

$$\begin{aligned}
 & \max F(\mathbf{x}) \\
 \text{subject to: } & Q^\alpha(\mathbf{x}) \leq Q_{\max}^\alpha \\
 & Q^\beta(\mathbf{x}) \geq Q_{\min}^\beta \\
 & S(\mathbf{x}) \leq L \\
 & \mathbf{x} \in A
 \end{aligned} \tag{1}$$

where \mathbf{x} is the decision vector for the adaptation actions to be performed, $F(\mathbf{x})$ is the objective function, $Q^\alpha(\mathbf{x})$ and $Q^\beta(\mathbf{x})$ are, respectively, those QoS attributes for which a max or min threshold is specified, $S(\mathbf{x})$ are the constraints on the offered load, and $\mathbf{x} \in A$ is a set of functional constraints on the \mathbf{x} value.

Within the MOSES framework, variations in the QoS requirements (e.g., new QoS thresholds specified by the users, or new utility objectives to be achieved) can be simply managed by a suitable re-instantiation of the optimization problem to be solved.

Problem Space Region Covered by MOSES. In summary, the colored boxes in Fig. 1 evidence the regions of the problem space resulting from the characterization outlined above that are covered by MOSES.

2.3 Benchmarking Criteria

Chap. 2 of this book (*Perpetual Assurances in Self-Adaptive Systems*) proposes some benchmarking criteria that can be used to compare different approaches for “perpetual assurances for self-adaptive systems”. The same chapter also presents in Sect. 2.3 a case study (Tele Assistance System (TAS)) that specifically refers to the SOA domain considered by MOSES. The TAS case study goal is to provide a set of specific challenges for that domain, which can be used as a testbed for the capabilities of a self-adaptation approach to fulfil requirements driven by the proposed benchmark criteria.

We defer to Sect. 5 a discussion of the compliance of MOSES with some of the benchmarking criteria. In the same section we also compare MOSES with other existing approaches according to the same criteria. In this section, instead, we briefly discuss how MOSES can tackle the challenges discussed for the TAS scenario (we refer to each challenge with the name used for it in Sect. 2.3 of Chap. 2 of this book (*Perpetual Assurances in Self-Adaptive Systems*), and refer to that chapter for its detailed description).

S1 - Individual service failure. Since concrete services implementing the Alarm Service may have different failure rates and costs, MOSES is able to select the single concrete service (or some redundancy pattern) that fulfils the

cost and reliability requirements. In particular, MOSES tackles this challenge by solving the following optimization problem received as input:

$$\begin{aligned} & \min \text{ cost}(AlarmService) \\ & \text{subject to: } \lambda(AlarmService) \leq X \end{aligned}$$

where $\lambda(s)$ denotes the failure rate of service s .

S2 - Variation of failure rate of services over time. MOSES deals with variation at runtime of the cost and/or failure rate of different concrete services for the Medical Analysis Service and Alarm Service thanks to its continuous monitoring activity, which can lead to a recalculation of a new solution for the optimization problem provided as input. In particular, analogously to case **S1**, this corresponds in the MOSES framework to solving a different instance of the optimization problem:

$$\begin{aligned} & \min \text{ cost}(AlarmService) + \text{cost}(MedicalAnalysisService) \\ & \text{subject to: } \lambda(AlarmService) + \lambda(MedicalAnalysisService) \leq X \end{aligned}$$

S3 - New service becomes available. Assuming that the newly arrived instance of Alarm Service is added to the pool of available services considered by MOSES, this addition is an event that triggers the recalculation of the optimization problem solution (which could possibly lead to selecting the newly arrived Alarm Service instance).

S4 - New type of service becomes available. Adding a new service type to an already existing workflow implies a replanning of the overall execution plan that was managed by the original workflow. Presently, MOSES is not designed to automatically deal with this kind of scenario. The new workflow must be built outside of the MOSES framework, and then provided to MOSES as a new input.

3 MOSES High-Level Architecture

Figure 2 shows the MOSES architecture, whose modules are organized according to the MAPE-K loop – *BPEL Engine*, *Composition Manager*, *Adaptation Manager*, *Optimization Engine*, *QoS Monitor*, *Execution Path Analyzer*, *WS Monitor*, *Service Manager*, *SLA Manager*, and *Data Access Library* –, and their interactions. A discussion about their implementation is presented in Sect. 4.

The *Execute* macro-component comprises the *Composition Manager*, *BPEL Engine*, and *Adaptation Manager* modules. The first module receives from the broker administrator the description of the composite service in some suitable workflow orchestration language (e.g., BPEL [33]), and builds a behavioral model of the composite service. To this end, the Composition Manager interacts with the Service Manager for the identification of the operations that implement the tasks required by the service composition. Once created, the system model is saved in the MOSES storage (i.e., the Knowledge macro-component) to make it accessible to the other system modules.

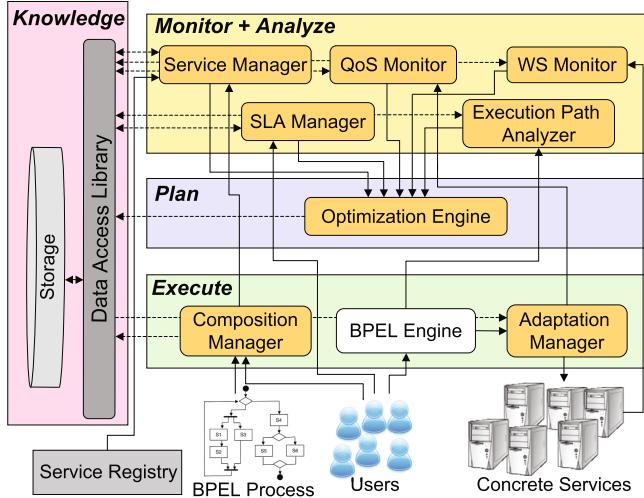


Fig. 2. MOSES high-level architecture.

While the Composition Manager is invoked rarely during the MOSES operativeness, the BPEL Engine and Adaptation Manager are the core modules for the execution and runtime adaptation of the composite service. The BPEL Engine is the software platform that actually executes the composite service, described in BPEL [33], and represents the user front-end for the composite service provisioning. It interacts with the Adaptation Manager to allow the invocation of the component services. The Adaptation Manager is the actuator at runtime of the adaptation actions. Indeed, for each operation invocation, it binds dynamically the request to the real endpoint(s) that represents the operation. This(these) endpoint(s) is(are) identified on the basis of the optimization problem solution determined by the Optimization Engine. The BPEL Engine and the Adaptation Manager also acquire raw data needed to determine respectively the usage profile of the composite service and the performance and reliability levels (specified in the SLAs) actually perceived by the users and offered by the concrete services. Together, the BPEL Engine and the Adaptation Manager are responsible for managing the user requests.

The *Optimization Engine* implements the *Plan* macro-component of the MAPE-K loop. It solves the optimization problem, which is based on the behavioral model initially built by the Composition Manager and instantiated with the parameters of the SLAs negotiated with both the MOSES users and the providers of the concrete services. The model is kept up to date by the monitoring activity carried out by the MOSES Monitor and Analyze macro-components. The solution of the optimization problem determines the adaptation policy in a given operating environment, which is saved on the MOSES Storage and retrieved by the Adaptation Manager for its actual implementation.

The modules in the *Monitor* and *Analyze* macro-components capture changes in the MOSES environment and, if they are relevant, modify at runtime the system model kept in the Storage layer and trigger the Optimization Engine to make it calculate a new adaptation policy. For each adaptation trigger mentioned in Sect. 2.2, we indicate here the corresponding MOSES module(s) responsible for tracking it: (i) the arrival/departure of a user with the associated SLA, possibly performing an admission control [1] (SLA Manager); (ii) observed variations in the SLA parameters of the constituent concrete services (QoS Monitor); (iii) addition/removal of an operation implementing a task of the abstract composition, the latter due either to graceful failures or crashes (Service Manager and WS Monitor); (iv) variations in the usage profile of the abstract tasks in the service composition (Execution Path Analyzer).

Finally, the *Knowledge* macro-component is accessed through the MOSES *Data Access Library* (MDAL), which allows to access the parameters describing the composite service and its operating environment (they include the set of tasks in the abstract composition, the corresponding candidate operations with their QoS attributes, and the current solution of the optimization problem that drives the composite service implementation).

4 MOSES Prototype

We have designed the MOSES prototype on the basis of the high-level architecture presented in the previous section and shown in Fig. 2. In this section, we first review the main features of the software prototype, which has been presented in [8, 12]; then, in Sect. 4.1 we present the relevant details regarding the implementation of the core MOSES modules and in Sect. 4.2 we discuss how MOSES can be extended to support a new service selection policy that may require the monitoring of additional QoS parameters. In Sect. 4.3 we discuss the overheads introduced by the adaptation policies and mechanisms implemented by MOSES. Finally, in Sect. 4.4 we briefly describe the evaluation tool we developed to test the MOSES performance and which is available within the MOSES package.

The MOSES prototype exploits the rich capabilities offered by the OpenESB framework [34] and the relational database MySQL, which both provide interesting features to enhance the scalability and reliability of complex systems. OpenESB, which was initially designed and developed under the direction of Sun Microsystems and is currently maintained by its own community, is a Java-based open source Enterprise Service Bus (ESB) that meets the requirements of SOA and provides a stable and lightweight JBI implementation. JBI, which stands for Java Business Integration, is considered as a philosophy for system integration, describing how to define and use a virtual bus to communicate between components. More precisely, it defines a messaging-based pluggable architecture and its major goal is to provide an enabling framework that facilitates the dynamic composition and deployment of loosely coupled participating applications and service-oriented integration components. The JBI specification defines as core components the Service Engines (SEs), the Binding Components (BCs),

and the Normalized Message Router (NMR). The SEs enable pluggable business logic and receive messages from the bus and send messages to the bus; the BCs enable pluggable external connectivity, being able to generate bus messages upon receipt of stimuli from an external source, or to generate an external action in response to a message received from the bus; the NMR directs normalized messages from source to destination components according to specified policies. Figure 3 illustrates the OpenESB-based architecture of MOSES.

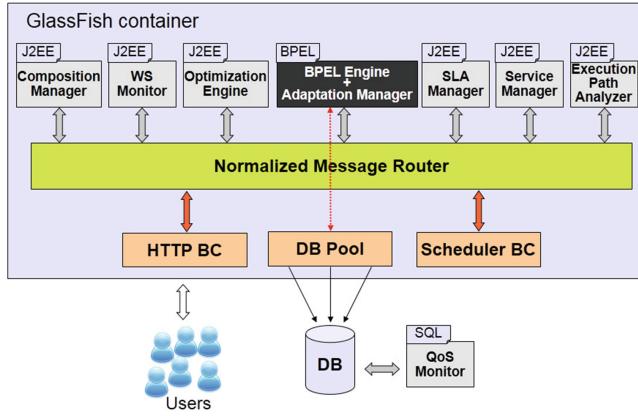


Fig. 3. MOSES OpenESB-based architecture.

Each MOSES component is executed by one SE, that can be either Sun BPEL Service Engine (a highly scalable orchestrator based on BPEL 2.0) for the business process logic, or J2EE Engine for the logic of the other MOSES modules. The resulting prototype has a good deployment flexibility, because each component can be accessed either as standard Web service or as EJB module through the NMR. However, to increase the prototype performance, we have exploited the NMR presence for all the inter-module communications, so that message exchanges are “in-process” and avoid to pass through the network protocol stack, as it would be for SOAP-based communications.

With regard to the MOSES storage layer, we rely on the relational database MySQL. However, to free the MOSES future developers from knowing the storage layer internals, we have developed a data access library, named *MOSES Data Access Library* (MDAL), that completely hides the data backend. This library currently implements a specific logic for MySQL, but its interface can be enhanced with other logics.

4.1 MOSES Modules

We now describe the implementation of the core MOSES modules for driving the adaptation decisions (Optimization Engine), executing at runtime the adaptation decisions (Adaptation Manager), monitoring the QoS attributes of the

concrete services offered by the third-party service providers (QoS Monitor and WS Monitor), managing the arrival and departure of users with the associated SLA (SLA Manager), and storing the knowledge needed by the MAPE-K control loop (storage layer). Since the remaining modules (Composition Manager, Service Manager, Execution Path Analyzer) that offer some useful but supplementary functionalities are not yet fully developed, they are currently not included in the MOSES distribution.

Optimization Engine. The Optimization Engine is the MOSES module that computes the solution of the optimization problem sketched in Sect. 2.2. Currently, MOSES supports two classes of optimization strategies for service selection (*per-flow* and *per-request*), corresponding to two different granularity levels.

At the per-request grain, the adaptation focuses on each single request submitted to the system, e.g., [3, 4, 10, 11, 25]; it aims at fulfilling the QoS constraints of that specific request, thus allowing potentially finer customization features. However, most per-request policies exhibit scalability and stability problems in a large scale system subject to a quite sustained flow of requests, because each request is managed independently of all the other concurrent ones [13].

The per-flow grain considers the flow of requests of a user rather than the single request, and the adaptation goal is to fulfill the QoS constraints that concern the global properties of that flow, e.g., the average composite service response time or its availability, e.g., [6, 12, 21]. However, adaptation policies adopting the per-flow grain are not able to ensure strict fulfillment of the required QoS attributes to each single request.

In MOSES we implemented three alternative optimization policies:

- the per-flow policy we presented in [12], where the service selection and the coordination pattern selection are jointly addressed with an efficient Linear Programming (LP) formulation;
- the per-request policy proposed by Ardagna and Pernici [4];
- the load-aware per-request we proposed in [13]; it relies on a Mixed Integer Linear Problem (MILP) formulation and exploits the multiple available implementations of each abstract task, thus realizing a runtime probabilistic binding that allows to achieve a randomized load balancing among the different concrete services available for the same functionality.

The Optimization Engine module is not self-contained as it relies on some external optimization software package to solve the optimization problem. In the current implementation, we use MATLAB® [24] for the per-flow class and CPLEX® [19] for the per-request class. In both optimization problem classes, the solution providing the current adaptation policy is stored into MOSES’s MySQL database through MDAL.

Adaptation Manager. The Adaptation Manager, which is responsible of the runtime binding of the abstract tasks to the corresponding implementations, is a

proxy module interposed between the BPEL process and the concrete services. It is called every time the BPEL Engine needs to invoke an external Web service. Differently from the other MOSES modules, it is not implemented as a JBI Service Engine. It is rather implemented as a standard Java class belonging to the application server classpath, and thus it is accessible by any application served by the application server itself. Its implementation offers a generic interface to the BPEL process so that it can be invoked with any kind of SOAP message as payload, wrapped in a proper envelope. Its task is to extract the SOAP message from the envelope and to drive it to the correct concrete service(s) according to the information received through the headers of the envelope and a pre-defined policy. To this end, the Adaptation Manager reads the most up-to-date adaptation policy plan from the database using the MDAL library, invokes the concrete service(s), and forwards the service response to the BPEL Engine.

We observe that the BPEL process needs to be customized in order to interact with the Adaptation Manager of MOSES. Such changes are all about replacing the invocation to concrete services with invocations to the Adaptation Manager and can be simply accomplished by following the instructions we provide with the MOSES documentation.

QoS Monitor. The QoS Monitor in MOSES employs a passive methodology to collect the actual values of QoS attributes provided by the concrete services, in particular response time and reliability. Data are collected without injecting additional load, but rather observing the system behavior in response to actual service invocations. Indeed, at each concrete service invocation, the perceived QoS is stored on a continuous time basis in the database using the MDAL library. Rather than using active probes that generate additional traffic on the monitored concrete services, we exploit passive collectors so to reduce the monitoring costs. However, this may result in a slower detection of SLA violations and can thus be appropriate when timeliness is not critical for the system [15].

The QoS Monitor is actually implemented as a MySQL stored procedure for performance reasons, being activated every time the Adaptation Manager invokes a concrete service. In the current MOSES implementation, we consider only the service response time as monitored QoS parameter; however, the extension to consider other QoS parameters is trivial.

To analyze the collected data regarding the response time, we use the online adaptive cumulative sum (Cusum) algorithm [30] for service response time monitoring and abrupt change detection. As discussed in [13], the online adaptive Cusum detector we implemented combines an Exponential Weighted Moving Average (EWMA) filter to track the slow varying response time series average with a two-sided Cusum test to detect abrupt changes in the series average which cannot be timely accounted by EWMA filter.

The Analyze phase of the MAPE-K control loop is completed by a detached thread of the Optimization Engine. Whenever the QoS Monitor detects that a SLA violation in the response time of some concrete service, it sets to true a flag stored in the database. Such a flag is periodically checked by the Optimization

Engine detached thread: if it is set to true, the Optimization Engine calculates the new adaptation policy using the updated model, where the perceived QoS values measured by the QoS Monitor replace those agreed in the SLA.

WS Monitor. The WS Monitor, which is implemented as a Web service and executed by a J2EE Service Engine, is configured to periodically probe all the concrete services known to MOSES in order to find out which services are currently available. Whenever the WS Monitor finds that some service changed its state (going from running to failed or vice-versa), it sends a trigger to the Optimization Engine. In its turn, the latter, using the updated model that reflects the service availability/unavailability, computes the new adaptation policy, that will be then executed by the Adaptation Manager.

SLA Manager. The SLA Manager performs a coarse-grain admission control in MOSES. Under the per-flow adaptation policy, it allows MOSES to decide whether to accept or reject a new potential user in such a way to guarantee non-functional QoS requirements to its already admitted users. Such admission control mechanism is required because the candidate concrete services, with whom the broker has defined a SLA, can be in a limited number with respect to the incoming request load and can thus be unable to provide the QoS levels required by the prospective users, as well as those of ongoing users already in the SOA system.

In the current MOSES implementation, we provide only a myopic admission control strategy, where the broker takes admission decisions using only the present system state, on the basis of the SLA of the requesting user and the SLAs of its currently admitted users. To this end, the SLA Manager invokes the Optimization Engine adding the requirements of the new potential user; if the instance of the per-flow optimization problem can be solved, then the contract will be established, otherwise it will be discarded. In [1] we proposed a forward-looking admission control policy based on Markov Decision Processes (MDP), with the goal to maximize the broker discounted reward while guaranteeing non-functional QoS requirements to its users. However, the MDP-based policy is not yet implemented into the MOSES prototype.

The SLA Manager allows also to free up resources when an existing contract expires; to this end, it invokes again the Optimization Engine to determine a new adaptation policy.

Storage Layer. The storage system behind MOSES is accessed through the MOSES Data Access Library (MDAL). Such module is of fundamental importance since it holds all the information needed to optimally drive the adaptation; therefore, it has been designed and implemented so to not constitute a bottleneck for the entire system. The goal of MDAL is to decouple the stored data from the programming interfaces used to access it: as a result, MOSES developers do not need to know how data are actually stored because they are hidden

behind a simple interface. Furthermore, the internals of the storage layer could be changed in the future without affecting other MOSES modules. Specifically, MDAL provides a set of entities that model the domain in which MOSES operates and a set of implemented interfaces to read, write, update, and delete the entities.

For the storage, MOSES relies on a relational database. Its ER diagram, shown in Fig. 4, includes as main entities: (i) **Process**, which represents the composite services deployed inside MOSES, providing a high-level description of the business processes and of their execution graphs; (ii) **User**, which represents the registered users; (iii) **AbstractService**, which represents the abstract part of the WSDL files and is complemented by the *AbstractOperation* weak entity; (iv) **ConcreteService**, which represents the actual services invoked by each business process, including its SLA parameters, and is complemented by the *ConcreteOperation* weak entity; (v) **SLA**, which represents the QoS classes offered for the corresponding composite service, including the QoS values agreed for each service class in the SLA and the actual perceived QoS values monitored during the MOSES execution; (vi) **Group**, which represents the coordination patterns (the currently supported ones are *single*, *alt*, and *par_or* [12]).

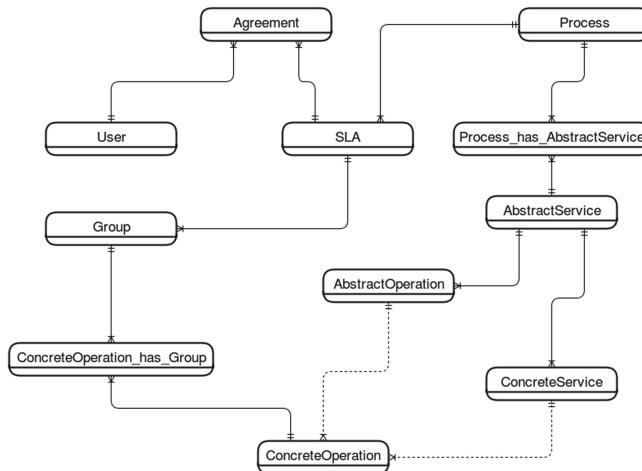


Fig. 4. Database ER diagram.

MOSES presently uses the relational database MySQL, which provides transactional features through the InnoDB storage engine and supports clustering and replication. We carefully optimized the MDAL methods to let the Adaptation Manager efficiently read the adaptation policy stored in the database, since this could happen even thousands of times per second. Since the solution of the optimization problem is scattered among multiple database tables, in order to avoid expensive and repeated table joins, MDAL creates a new **solution** table from

temporary in-memory tables that are then dropped whenever a new optimal solution is computed. Furthermore, we improved the performance of executing queries to the database by exploiting the connection pool mechanism provided by Glassfish, which allows to cache database connections so that they can be reused when future requests to the database are required.

4.2 MOSES Extensions

MOSES has a modular and easily extendible architecture. It has been realized in such a way that the prospective developers could both easily change the behaviour of already developed components and add new currently unplanned functionalities. For instance, in order to develop a new service selection policy that uses the QoS parameters already supported by MOSES, the developer has just to implement with his/her own code the Optimization Engine Java interface and change the proper MOSES configuration file. The new class will be dynamically loaded at runtime. Such an extension does not require any modification to the other MOSES modules.

The addition of new functionalities may require instead to change the data model. For instance, if the prospective developer is interested in considering a service selection based on the actual network latency between MOSES and the concrete services, s/he must add that QoS attribute to the data model. Such a change must be then reflected into MDAL and a new MOSES module must be developed to make MOSES able to measure the new metric. The newly developed module can then be easily plugged into the architecture presented in Fig. 3, where it will be able to exploit other services, first of all the Optimization Engine. The latter must be also properly extended to handle the new QoS attribute. We observe that the modifications to MDAL needed for the new module and for the extended Optimization Engine do not impact the normal operativeness of all the other modules, which can still use the original version of the data access library.

Details on how to extend MOSES can be found in the documentation available at <http://www.ce.uniroma2.it/moses/>.

4.3 MOSES Overheads

As observed in Chap. 2 of this book (*Perpetual Assurances in Self-Adaptive Systems*), to assure compliance with the requirements for self-adaptive systems at runtime, the employed techniques must not interfere with the ability of the self-adaptive system to deliver its intended functionality effectively and efficiently. To achieve this goal, in the MOSES design and implementation we took care of the different types of overheads introduced by the runtime adaptation management. They can be classified according to the MOSES macro-components: (i) overhead due to the Plan macro-component; (ii) overhead of the Execution macro-component due to the runtime binding of the task endpoints to concrete implementations; (iii) overhead due to the Monitor and Analyze macro-components.

Plan Overhead. The overhead introduced by the Optimization Engine depends on the time taken to calculate a new adaptation policy (and hence the class of optimization problem formulation) as well as on the synchronous or asynchronous invocation of the Optimization Engine with respect to the service execution flow, as illustrated in Fig. 5.

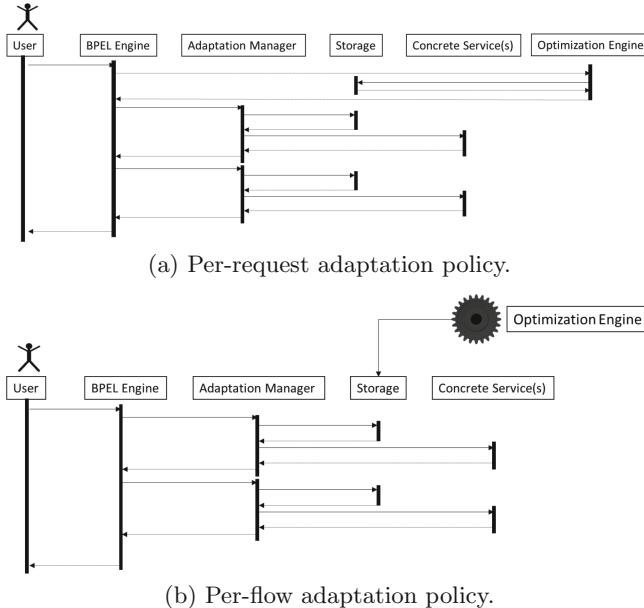


Fig. 5. Interaction among the MOSES modules in the service execution flow.

As regards the optimization problem formulation, the overhead depends on the computational complexity of the optimization problem that drives the adaptation policy. We demonstrated in [12] that the adoption of a linear programming model as optimization problem helps considerably in terms of scalability with respect to other approaches in literature that rely instead on more computationally expensive formulations: specifically, our approach results from one to two orders of magnitude faster. For example, for problem instances of reasonably large size (1000 service tasks in the abstract composite service and 50 concrete services implementing each task), the LP problem formulation adopted by MOSES requires 8.64 s to be solved, against 451.30 s of the per-flow approach in [6] and 19.88 s of the per-request approach in [4] (see Table 5 in [12] for the complete experiment). Nevertheless, the exploitation of redundancy patterns can result in excessive computational costs in case of large scale problem instances where the number of candidate services grow dramatically. Such costs can be restrained by either limiting the use of these patterns to a subset of the abstract tasks, or by limiting the maximal number of candidate services that can be used

to implement a redundancy pattern. To this end, the redundancy degree is a tunable parameter within the per-flow policy of the Optimization Engine.

As regards the synchrony of the optimization problem solution and the service execution flow, we observe that the per-request adaptation policy requires to solve the optimization problem synchronously to each service request (see Fig. 5a), because no QoS class is defined a-priori and the required QoS attributes are set by the user for each request addressed to the broker. After having solved the optimization problem, the corresponding adaptation policy is stored in the database and then used for the subsequent invocations pertaining to the same user request.

On the contrary, in the per-flow adaptation policy, the optimization problem is solved asynchronously with respect to the service execution flow (see Fig. 5b), while incoming service requests are served by the Adaptation Manager according to the previously calculated policy, which is stored in the database. Therefore, the computational cost of the optimization problem does not directly impact on the MOSES ability to respond to the user requests (because in the meanwhile the old, sub-optimal policy can be used), but only affects its responsiveness in updating the adaptation policy.

Execute Overhead. The overhead in the Execute phase is related to the runtime binding of abstract tasks with concrete services carried out by the Adaptation Manager and affects each request to the composite service as many times as the number of `invoke` activities executed in the BPEL process, as shown in Fig. 5. For every invocation of an abstract task, the Adaptation Manager, which is stateless, retrieves the current adaptation policy kept in the storage layer and, according to it, determines the actual operation(s) (and possibly the coordination pattern) to implement the abstract task.

We measured the overhead added by the MOSES runtime binding to the execution of the GlassFish ESB engine, finding that the MOSES response time is on average 74% higher than that provided by the plain BPEL engine [12].

Monitor and Analyze Overhead. We observe that the Analyze phase does not affect the overall service time perceived by a user, since the related operations are executed asynchronously with respect to the composite service.

The most time consuming and frequent monitoring activity is that performed with respect to the QoS parameters (specifically, the response time) offered by the concrete services. In this case, the MOSES monitoring overhead is about 1 ms for each `invoke` activity in the composite service, as it only involves inserting the operation response time in a database table: for each `invoke` activity, MOSES gets the timestamp before and after the invocation itself, calculates the observed response time and then puts it into the storage layer.

4.4 MOSES Evaluation Tool

To issue requests to the composite service managed by MOSES and to mimic the behavior of users that establish SLAs before accessing the service, we implemented a workload generator in C language using the Pthreads library.

The workload generator was designed to test the per-flow adaptation policy but can be as well as used for the per-request adaptation policies, as we did in [13]. It is based on an open system model, where users requesting a given service class k offered by MOSES arrive at mean *user inter-arrival rate* Λ_k . Each class k user u is characterized by its SLA parameters and by the *contract duration* t_u^k . Each incoming user is subject to an admission control, carried out by the SLA Manager as discussed in Sect. 4.1. If the SLA Manager admits the user, it starts generating requests to the composite service according to the rate λ_u^k until its contract ends. Otherwise, the user terminates.

In the workload model of our generator we assume exponential distributions of parameters Λ_k and $1/t_k$ for the user inter-arrival time and contract duration, respectively. We also assume that the request inter-arrival rate and the operations service time follow a Gaussian distribution. Such classic distributions can be easily changed in case new studies will evidence some peculiar characteristics of SOA traffic. The workload generator uses multiple independent random number streams for each stochastic component of the workload model. The generator reports as metrics for the composite service managed by MOSES the QoS attributes (i.e., response time, reliability, and cost) that are currently considered by MOSES.

To experimentally evaluate the performance of the self-adaption policies, we set up a composite service that mimics a travel planner [12]. The concrete services that we provide in the MOSES package are simple stubs, without internal logic; however, their non-functional behavior can be easily set to conform to the guaranteed levels expressed in their SLA.

5 Related Work

Several solutions have been proposed for the self-adaptation of SOA systems, mainly focusing on the runtime service selection (e.g., [3–6, 10, 14, 16, 17, 22, 23, 32, 37, 38]). Service selection has been and still is a challenging problem because the number of candidate services offering the same functionalities but differing in QoS is increasing with the prevalence of service-based systems and Cloud computing. However, only few frameworks and platforms have been designed, implemented and evaluated in the scope of self-adaptation of SOA systems. To the best of our knowledge, the platforms that share some common features with MOSES include MUSIC [35], SASSY [26, 28], VieDAME [31], DISCoRSO [5], and VRESCo [29]. In this section we review their characteristics and features according to the benchmarking criteria proposed in Chap. 2 of this book (*Perpetual Assurances in Self-Adaptive Systems*), which are summarized in Table 1.

MOSES handles requirement variability by solving an optimization problem, aimed at maximizing a utility function subject to constraints given by the current operating environment. It implements two classes of optimization problems, named *per-flow* and *per-request* as described in Sect. 4.1. In the per-flow scenario the application architect is expected to define the QoS classes that MOSES will advertise and offer to perspective users. Conversely, in the per-request scenario,

Table 1. Summary of assurance characteristics of existing frameworks

Benchmark aspect	Criteria	MOSES	MUSIC	SASSY	VieDAME	DISCoRSO
	Variability	✓	✓	✓	✓	✓
	Inaccuracy & incompleteness	✓	✓	✓	✓	✓
Capabilities of approaches to provide assurances	Conflicting criteria	✓	✓	✓	✓	✓
	User Interaction		✓			
	Handling alternatives	✓	✓	✓	✓	✓
Basis of assurance benchmarking	Historical data	✓	✓	✓	✓	✓
	Projections in the future					✓
	Human evidence		✓			
Stringency of assurances	Assurance rational	✓				✓
Performance of approaches	Timeliness	✓				
	Computational overhead	✓		✓	✓	
	Complexity	✓		✓		✓

no QoS class is defined a-priori and the required QoS attributes must be set by the user for each request addressed to the broker. As a consequence, in the former scenario MOSES can compute a-priori an optimal adaptation strategy for each service class defined by the application architect and subsequent re-optimizations are only due to changes in the environment or to user arrival/departure. In case the user tries to establish a service contract when MOSES has not sufficient available resources, the optimization problem will result unfeasible, thus leading to a rejection of the new contract. On the other hand, in the per-request scenario MOSES has to compute a new adaptation policy for each request it receives, being each request characterized by its own QoS requirements. Should the user require too stringent QoS attributes, the optimization problem will be unfeasible, thus leading to the request rejection. Conflicting optimization goals are managed by MOSES using the SAW technique in the utility function. Since MOSES is not designed for long lasting processes, user interaction is not expected at runtime, but adaptation policy preemption is possible in the per-flow adaptation strategy. MOSES continuously monitors service execution time and reliability. The collected data is then analyzed to discover possible SLA violations and therefore to trigger a new optimization request. MOSES provides strict QoS assurances in case of the per-request adaptation strategy, while it is capable of assuring average QoS attributes on a flow of request with the per-flow adaptation strategy. From the performance perspective, MOSES is able to provide a runtime dynamic binding with a very small overhead (more details are provided in Sect. 4.3), while the optimization process, in the per-flow arrangement, is designed as a LP problem,

which can be solved in polynomial time. The per-request optimization problems are formulated as MILP problem, thus having a NP-hard complexity.

MUSIC handles requirements variability by adapting the applications it manages according to the current context (e.g., the availability of certain resources such as GPS signal and WiFi networks). Inaccuracy and incompleteness at design time of what will be the runtime environment are managed through a constant monitoring activity: MUSIC is able to detect the health state of known services and, should any of them not being compliant with the negotiated QoS levels, it can be replaced by some other known service. Furthermore, MUSIC can discover new services and integrate them seamlessly in the service composition logic. Differently from the other considered platforms, MUSIC is able to concurrently manage several applications on a single device (e.g., a smartphone). Therefore, given the inherent constraints of the device it runs on, it must be able to maximize a utility function that addresses all the QoS parameters of every running application, trying to provide a weighted performance compromise among them. For this behavior to be effective, the user needs to interact with the framework in order to choose, for instance, the service class for a given application. Both human evidences and monitored historical performance data are used to determine whether to compute a new adaptation strategy. The framework, however, is not thought for mission-critical applications and there is no assurance it will run with an optimal configuration, because the authors did not formulate any optimization problem. Rather, the best possible configuration is chosen from a set of possible configurations after evaluating a utility function; however, how to populate the set of possible configurations is not discussed.

SASSY is a model-driven framework that provides runtime adaptation of service compositions in response to system variations as those caused by services violating the negotiated SLA. It adopts a MAPE-K loop to monitor the exploited services, analyze monitored data, plan a new implementation of the application, and finally execute the application. SASSY allows the application architect to specify at design time the QoS attributes desired for the application, which are then used to build a runtime model that is then exploited to achieve the QoS goals. Possibly conflicting criteria are managed by weighting a utility function and optionally adding constraints like cost. User interaction is not expected at runtime: the entire framework is designed to let the application architects specify their QoS constraints at design time. However, given such QoS constraints, new alternatives are automatically evaluated in response to changing operating conditions (e.g., new services are discovered or existing services violate their SLA). The reactive behavior of SASSY is backed by an analysis of data regarding the performance of the exploited services. Since the formulated optimal service selection problem is NP-hard, a heuristic based on hill-climbing was proposed to solve it efficiently. As a consequence, there is no guarantee that the actual instantiated architecture will strictly satisfy the required QoS.

VieDAME is able to handle variability: it decouples the required functionalities from the actual concrete services and provides a runtime dynamic binding. The selected concrete services are then continuously monitored in order to verify

whether they satisfy the agreed SLA. Should VieDAME find some violation, the bound implementations can be replaced, according to the specific service selector used to implement the dynamic binding. Furthermore, VieDAME addresses the problem of carefully designing domain-specific QoS requirements by providing a simple language (named VieDASSL) that can be exploited by domain experts to define QoS requirements. VieDAME service selection is based on a score computed on the basis of some service selection rules named *Factor rules*: the simplest is the *Simple Factor Rule*, which is able to manage at most one QoS attribute at a time, thus not providing a tool to manage conflicting criteria. The latter are addressed with the *Sum Selection Rule*, which provides to the domain expert a tool to describe a weighted sum of QoS attributes for the implementation of a given functionality. Alternative services are considered for two purposes: load balancing and SLA violations. VieDAME is built on top of a MAPE-K loop for autonomic systems which continuously monitors the perceived QoS attributes of every concrete service involved and, based on historical data, it determines whether to replace the services. VieDAME does not support strict fulfillment of QoS attributes: it exploits VieDASSL to define rules that are then used to drive the computation of scores for the available services. Such scores are finally used to build a classification of the services and the best one is chosen to implement the required functionality. To avoid overloading the best service, a selection post-processor is also implemented: instead of always using the same best concrete service, a subset of the available services with a score greater than a threshold is considered. The CPU overhead introduced by the VieDAME runtime binding and monitoring is in the range of 10%–20% with respect to the execution of the same BPEL process without self-adaptation features.

DISCoRSO lets the application architects define both local and global QoS constraints. During the application execution, QoS attributes are continuously monitored in order to detect possible SLA violations and re-optimize the whole execution process. To this end, monitored data is stored into the database and then used by the analyzer module which is the responsible of SLA violation detections. DISCoRSO uses a mix of historical data analysis and projections in the future to minimize the probability of not satisfying the SLA of the whole application, also by eventually triggering a pre-emptive live re-optimization. This behavior is particularly useful to provide QoS assurances on long-term executions. The optimization problem is formulated as a MILP problem (thus having NP-hard complexity) and supports the management of possibly conflicting criteria. Its optimal solution, which must be computed for every invocation to the composite application, supports a strict fulfillment of the required QoS attributes.

VRESCo neither proposes nor implements optimal or sub-optimal solutions to provide an adaptation strategy: it rather provides a framework for dynamic binding and a query language that can be used to retrieve execution plans that must be afterwards processed by an optimization algorithm. Therefore, VRESCo cannot provide by itself assurances on QoS attributes, rather it provides an execution platform which can be enriched by adding optimization features. Specifically, referring to MAPE-K loop, it only implements the Execute phase. However, VRESCo is the only considered platform which explicitly supports

Service Mediation: it accepts from the user a high-level representation of the data that will be used as input for the concrete services. These data are then lowered (i.e., transformed from high-level representation into low-level format) in order to be compatible with the given concrete service. The response is instead lifted (i.e., transformed from low-level format to high-level representation) in order to be compatible with user expectations. From a performance perspective, the VRESCo runtime binding adds about 400 ms to the service execution time, without considering the service mediation. This additional time is due to the addition of a proxy which, for each service invocation, queries the database in order to know which service to invoke.

6 Conclusions

We have presented MOSES, a software platform supporting QoS-driven adaptation of service-oriented systems. MOSES is architected as a broker that controls the self-adaptation of a composite service by implementing the functionalities of a MAPE-K control loop. The modular MOSES architecture facilitates the integration in the overall platform of different adaptation policies and mechanisms, thus making it a suitable testbed for their experimentation, thanks also to the availability of the complete platform source code.

As described in the previous sections, MOSES provides a complete implementation of the core functionalities of the MAPE-K loop, tailored for the SOA environment. However, we remark that some useful functionalities, which would make MOSES a comprehensive solution for the management of SOA systems, are currently not included in the MOSES distribution. They include, in particular: (i) the negotiation of SLAs with the prospective users of the managed composite service, and with the providers of services exploited by MOSES; (ii) the discovery of services to be included in the pool of possible candidates for the composition of the managed service; (iii) the automatic derivation of the optimization problem that constitutes the core of the MOSES strategy, from a model of the composite service, QoS requirements and constraints.

The QoS-driven self-adaptation policies provided by MOSES suffer from two limitations: first, they do not consider the network latency and bandwidth between the broker and the concrete services; second, they use point estimates of the QoS attributes of the concrete services which may not correctly reflect the actual statistics, e.g., under high variance the point estimates may take too long to converge to the actual value. We plan to address the first issue by including network parameters within the MOSES model and devising a new formulation of the optimization problem. We will tackle the second issue by adopting robust optimization techniques [9] that are capable to provide solutions which satisfy the composite service QoS requirements despite the QoS attributes uncertainty and/or variability.

As a final remark, we note that MOSES itself is a complex system that could benefit from the addition of self-adaptation features to automatically scale in/out in response to variations in the load of user requests it has to manage, or to adjust its reliability. This extension of the MOSES platform is subject of ongoing work.

Acknowledgments. V. Cardellini and F. Lo Presti acknowledge the support of ICT COST Action IC1304 ACROSS.

References

1. Abundo, M., Cardellini, V., Lo Presti, F.: Admission control policies for a multi-class QoS-aware service oriented architecture. *ACM SIGMETRICS Perform. Eval. Rev.* **39**(4), 89–98 (2012)
2. Alférez, G., Pelechano, V., Mazo, R., Salinesi, C., Diaz, D.: Dynamic adaptation of service compositions with variability models. *J. Syst. Softw.* **91**, 24–47 (2014)
3. Alrifai, M., Risso, T.: Combining global optimization with local selection for efficient QoS-aware service composition. In: Proceedings of WWW 2009, pp. 881–890 (2009)
4. Ardagna, D., Pernici, B.: Adaptive service composition in flexible processes. *IEEE Trans. Softw. Eng.* **33**(6), 369–384 (2007)
5. Ardagna, D., Baresi, L., Comai, S., Comuzzi, M., Pernici, B.: A service-based framework for flexible business processes. *IEEE Softw.* **28**(2), 61–67 (2011)
6. Ardagna, D., Mirandola, R.: Per-flow optimal service selection for web services based processes. *J. Syst. Softw.* **83**(8), 1512–1523 (2010)
7. Baresi, L., Di Nitto, E., Ghezzi, C.: Toward open-world software: issue and challenges. *IEEE Comput.* **39**(10), 36–43 (2006)
8. Bellucci, A., Cardellini, V., Di Valerio, V., Iannucci, S.: A scalable and highly available brokering service for SLA-based composite services. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 527–541. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17358-5_36
9. Ben-Tal, A., El Ghaoui, L., Nemirovski, A.: Robust Optimization. Princeton University Press, Princeton (2009)
10. Berbner, R., Spahn, M., Repp, N., Heckmann, O., Steinmetz, R.: Heuristics for QoS-aware web service composition. In: Proceedings of IEEE ICWS 2006, pp. 72–82 (2006)
11. Canfora, G., Di Penta, M., Esposito, R., Villani, M.: A framework for QoS-aware binding and re-binding of composite web services. *J. Syst. Softw.* **81**(10), 1754–1769 (2008)
12. Cardellini, V., Casalicchio, E., Grassi, V., Iannucci, S., Lo Presti, F., Mirandola, F.: MOSES: a framework for QoS driven runtime adaptation of service-oriented systems. *IEEE Trans. Softw. Eng.* **38**(5), 1138–1159 (2012)
13. Cardellini, V., Di Valerio, V., Grassi, V., Iannucci, S., Lo Presti, F.: QoS driven per-request load-aware service selection in service oriented architectures. *Int. J. Softw. Inform.* **7**(2), 195–220 (2013)
14. Chen, Y., Huang, J., Lin, C.: Partial selection: an efficient approach for QoS-aware web service composition. In: Proceedings of IEEE ICWS 2014 (2014)
15. Erradi, A., Maheshwari, P., Tosic, V.: WS-policy based monitoring of composite web services. In: Proceedings of ECOWS 2007, pp. 99–108. IEEE (2007)
16. He, Q., Yan, J., Jin, H., Yang, Y.: Quality-aware service selection for service-based systems based on iterative multi-attribute combinatorial auction. *IEEE Trans. Softw. Eng.* **40**(2), 192–215 (2014)
17. Huang, J., Liu, G., Duan, Q., Yan, Y.: Qos-aware service composition for converged network-cloud service provisioning. In: Proceedings of IEEE SCC 2014, pp. 67–74 (2014)

18. Hwang, C., Yoon, K.: Multiple Criteria Decision Making. Lecture Notes in Economics and Mathematical Systems, vol. 186. Springer, Heidelberg (1981). <https://doi.org/10.1007/978-3-642-48318-9>
19. IBM Corp.: CPLEX Optimizer (2016). <http://www.ibm.com/software/commerce/optimization/cplex-optimizer/>
20. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. *IEEE Comput.* **36**(1), 41–50 (2003)
21. Klein, A., Ishikawa, F., Honiden, S.: Efficient QoS-aware service composition with a probabilistic service selection policy. In: Maglio, P.P., Weske, M., Yang, J., Fanti-nato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 182–196. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17358-5_13
22. Klein, A., Ishikawa, F., Honiden, S.: SanGA: a self-adaptive network-aware approach to service composition. *IEEE Trans. Serv. Comput.* **7**(3), 452–464 (2014)
23. Leitner, P., Hummer, W., Dustdar, S.: Cost-based optimization of service compositions. *IEEE Trans. Serv. Comput.* **6**(2), 239–251 (2013)
24. MathWorks Inc.: MATLAB® (2016). <http://www.mathworks.com/>
25. Menascé, D., Casalicchio, E., Dubey, V.: On optimal service selection in service oriented architectures. *Perform. Eval.* **67**(8), 659–675 (2010)
26. Menascé, D., Gomaa, H., Malek, S., Sousa, J.P.: Sassy: a framework for self-architecting service-oriented systems. *IEEE Softw.* **28**(6), 78–85 (2011)
27. Menascé, D.A.: Qos issues in web services. *IEEE Internet Comp.* **6**(6), 72–75 (2002)
28. Menascé, D.A., Ewing, J.M., Gomaa, H., Malek, S., Sousa, J.P.: A framework for utility-based service oriented design in SASSY. In: Proceedings of WOSP/SIPEW 2010, pp. 27–36. ACM (2010)
29. Michlmayr, A., Rosenberg, F., Leitner, P., Dustdar, S.: End-to-end support for QoS-aware service selection, binding, and mediation in VRESCo. *IEEE Trans. Serv. Comput.* **3**(3), 193–205 (2010)
30. Montgomery, D.: Introduction to Statistical Quality Control. Wiley, Hoboken (2008)
31. Moser, O., Rosenberg, F., Dustdar, S.: Domain-specific service selection for composite services. *IEEE Trans. Softw. Eng.* **38**(4), 828–843 (2012)
32. Ngoko, Y., Goldman, A., Milojicic, D.: Service selection in web service compositions optimizing energy consumption and service response time. *J. Internet Serv. Appl.* **4**(1), 19 (2013)
33. OASIS: Web Services Business Process Execution Language Version 2.0 (2007)
34. OpenESB - The Open Enterprise Service Bus (2015). <http://www.open-esb.net/>
35. Rouvoy, R., et al.: MUSIC: middleware support for self-adaptation in ubiquitous and service-oriented environments. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 164–182. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_9
36. Salehie, M., Tahvildari, L.: Self-adaptive software: landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.* **4**(2), 1–42 (2009)
37. Schuller, D., Siebenhaar, M., Hans, R., Wenge, O., Steinmetz, R., Schulte, S.: Towards heuristic optimization of complex service-based workflows for stochastic QoS attributes. In: Proceedings of IEEE ICWS 2014, pp. 361–368 (2014)
38. Zivkovic, M., Bosman, J., van den Berg, H., van der Mei, R., Meeuwissen, H., Nunez-Queija, R.: Run-time revenue maximization for composite web services with response time commitments. In: Proceedings of IEEE AINA 2012, pp. 589–596 (2012)

Author Index

- Anders, Gerrit 188
Andersson, Jesper 3, 64
Autili, Marco 282
- Baresi, Luciano 3, 64
Bencomo, Nelly 3, 31
Braberman, Victor 377
Brun, Yuriy 3, 64
- Calinescu, Radu 3, 31, 223
Cámera, Javier 3, 31, 154
Cardellini, Valeria 409
Cardozo, Nicolás 307
Casalicchio, Emiliano 409
Clarke, Siobhán 307
Cohen, Myra B. 3, 64
Collet, Philippe 251
- D'Ippolito, Nicolas 377
de Lemos, Rogério 3
- Eberhardinger, Benedikt 188
- Garlan, David 3, 154
Gerasimou, Simos 223
Ghezzi, Carlo 3, 31
Giese, Holger 3, 90
Gorla, Alessandra 3, 64
Grassi, Vincenzo 3, 31, 409
Grunské, Lars 3, 31
- Iannucci, Stefano 409
Inverardi, Paola 3, 31, 282
- Jezequel, Jean-Marc 3, 31
Johnson, Kenneth 223
- Knapp, Alexander 188
Kramer, Jeff 377
Kříkava, Filip 251
- Litoiu, Marin 3, 90
Lo Presti, Francesco 409
- Malek, Sam 3, 31
Marchand, Nicolas 349
Mens, Kim 307
Metzger, Andreas 137
Mirandola, Raffaela 3, 31, 409
Moreno, Gabriel A. 154
Mori, Marco 3, 31
Müller, Hausi A. 3, 90
- Paterson, Colin 223
Perucci, Alexander 282
- Reif, Wolfgang 188
Rouvoy, Romain 3, 90, 251
Rubira, Cecília M. F. 3, 64
Rutten, Eric 3, 90, 349
- Schmerl, Bradley 3, 64, 154
Seebach, Hella 188
Seinturier, Lionel 251
Sharifloo, Amir Molzam 137
Shaw, Mary 3, 90
Siefert, Florian 188
Simon, Daniel 349
Sykes, Daniel 377
- Tamburrelli, Giordano 3, 31
Tamura, Gabriel 3, 90
Tivoli, Massimo 282
- Uchitel, Sebastian 377
- Villegas, Norha M. 3, 90
Vogel, Thomas 3, 64
- Weyns, Danny 3, 31
- Zambonelli, Franco 3, 64