Lecture Notes in Computer Science 7475

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Rogério de Lemos Holger Giese
Hausi A. Müller Mary Shaw (Eds.)

Software Engineering for Self-Adaptive Systems II

International Seminar
Dagstuhl Castle, Germany, October 24-29, 2010
Revised Selected and Invited Papers

Springer

Volume Editors

Rogério de Lemos
University of Kent, School of Computing
Canterbury, Kent CT2 7NF, UK
and
Centre for Informatics and Systems
of the University of Coimbra (CISUC)
3030-290 Coimbra, Portugal
E-mail: r.delemos@kent.ac.uk

Holger Giese
University of Potsdam
Hasso Plattner Institute for Software Systems Engineering
Postfach 900460, 14440 Potsdam, Germany
E-mail: holger.giese@hpi.uni-potsdam.de

Hausi A. Müller
University of Victoria, Department of Computer Science
STN CSC, Victoria, BC, V8W 3P6, Canada
E-mail: hausi@cs.uvic.ca

Mary Shaw
Carnegie Mellon University, School of Computer Science
5000 Forbes Avenue, Pittsburgh, PA 15213-3891, USA
E-mail: mary.shaw@cs.cmu.edu

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-35812-8 e-ISBN 978-3-642-35813-5
DOI 10.1007/978-3-642-35813-5
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012954418

CR Subject Classification (1998): D.2, D.3, F.1.1, I.2.2

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

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

Preface

The complexity of current software-based systems has led the software engineering community to look for inspiration in diversely related fields (e.g., robotics and control theory) as well as other areas (e.g., biology) to find innovative approaches for building, running, and managing software systems and services. Therefore, *self-adaptation*—systems that are able to adjust their behavior or structure at run-time in response to their perception of the environment and the system itself – has become a hot topic within the software engineering community.

This book is one of the outcomes of Dagstuhl Seminar 10431 on "Software Engineering for Self-Adaptive Systems" held in October 2010. It is the second book in the series and comprises a research roadmap, four working group papers, and invited papers from recognized experts in the field. The research roadmap, complemented by four group papers that detail the issues covered by the roadmap, summarizes the Dagstuhl Seminar discussions and provides insights into key features of self-adaptive software systems. All the papers in this book are peer-reviewed, with the exception of the roadmap 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 Roadmap," "Requirements and Policies," "Design Issues," and "Applications."

Part one of the book, which is entitled "Research Roadmap," includes the roadmap paper "Software Engineering for Self-Adaptive Systems: A Second research Roadmap" that indentifies selected software engineering research challenges for self-adaptive systems, and the four working group papers that elaborate the four self-adaptation perspectives presented in the roadmap paper. Instead of dealing with a wide range of topics associated with the field, this roadmap paper focuses on four fundamental topics of self-adaptation: design space of adaptive solutions, towards software engineering processes for self-adaptive systems, from centralized to decentralized control, and practical run-time verification and validation for self-adaptive systems.

The second paper in the first part of this book by Brun, Desmarais, Geihs, Litoiu, Lopes, Shaw, and Smit, entitled "A Design Space for Self-Adaptive Systems," discusses the importance of systematic design, identifies the dimensions of the self-adaptive system design space, and identifies key design decisions, questions, and possible answers relevant to the design space, which are organized into five clusters: observation, representation, control, identification, and enacting adaptation.

The third paper entitled "Software Engineering Processes for Self-Adaptive Systems," authored by Andersson, Baresi, Bencomo, de Lemos, Gorla, Inverardi, and Vogel, argues that traditional software engineering processes need to be reconceptualized to distinguish between development-time and run-time activities, and to support designers in taking decisions on how to engineer such systems properly. The paper also identifies a number of challenges on re-conceptualization and proposes initial ideas based on process modeling.

The fourth paper "On Patterns for Decentralized Control in Self-Adaptive Systems," authored by Weyns, Schmerl, Grassi, Malek, Mirandola, Prehofer, Wuttke, Andersson, Giese, and Göschka, aims to consolidate knowledge on decentralized control in self-adaptive systems in the form of patterns of interacting MAPE loops to describe the different types of control in self-adaptive systems.

The fifth paper in the first part of this book by Tamura, Villegas, Müller, Sousa, Becker, Karsai, Mankovskii, Pezzè, Schäfer, Tahvildari, and Wong, entitled "Towards Practical Runtime Verification and Validation of Self-Adaptive Software System," analyzes fundamental challenges and concerns in the development of verification and validation (V&V) methods and techniques that provide certifiable trust in self-adaptive and self-managing systems and presents a proposal for including V&V operations explicitly in feedback loops for ensuring the achievement of software self-adaptation goals.

Part two of this book, entitled "Requirements and Policies," consists of three papers describing approaches in which requirements and policies assume a central role in the development, deployment, and operation of self-adaptive software systems.

The first paper by Souza, Lapouchnian, Robinson, and Mylopoulos, entitled "Awareness Requirements for Adaptive Systems," discusses awareness requirements, which are characterized syntactically as requirements that refer to other requirements or domain assumptions and their success or failure at run-time. It presents how awareness requirements are monitored, and provides a discussion on how to go from awareness requirements to self-adaptive systems. The proposed approach has been evaluated by analyzing, designing, and developing a simulation of a real-world system—an ambulance dispatch system.

The second paper, entitled "Self-Management of Distributed Systems Using High-Level Goal Policies," by Rosa, Rodrigues, and Lopes describes an approach to automate the selection of the adaptations that should be performed in response to changes in the execution environment. The approach identifies key aspects to describe goals, specify adaptations, and select adaptations when addressing distributed components, and discusses how to perform the system monitoring and execution of the adaptations in a distributed setting.

The last paper of this part by Ghezzi and Sharifloo, entitled "Dealing with Non-Functional Requirements for Adaptive Systems via Dynamic Software Product-Lines," presents an approach for ensuring continuous satisfaction of non-functional requirements using self-adaptation. To achieve this, the authors propose that the implementation should be architected as a dynamic software product line (DSPL), whose target configurations can be generated dynamically. They also discuss how the DSPL can be verified against non-functional requirements at development-time through model checking.

Part three of the book covers "Design Issues" and includes three papers on design perspectives in self-adaptive software systems.

The first paper by Esfahani and Malek, entitled "Uncertainty in Self-Adaptive Software Systems," characterizes the sources of uncertainty in a self-adaptive software system and demonstrates its impact on the system's ability to satisfy its objectives. It also provides an alternative notion of optimality that explicitly incorporates the uncertainty underlying the knowledge (models) used for decision making.

The second paper, entitled "A Software Life Cycle Process to Support Consistent Evolutions," by Inverardi and Mori defines a generic model-centric software life cycle process for context-aware self-adaptive systems. The proposed process supports concrete mechanisms to achieve consistent evolution at development-time and run-time through static and dynamic decision-making procedures.

The third paper on design issues, entitled "DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems," by Villegas, Tamura, Müller, Duchien, and Casallas, introduces a reference model for engineering highly dynamic adaptive software systems that helps guarantee the coherence of adaptation mechanisms with respect to changes in adaptation goals, and monitoring mechanisms with respect to changes in both adaptation goals and adaptation mechanisms.

Part four of the book contains four papers covering a wide range of "Applications."

The first paper in this part by Dubey, Karsai, and Mahadevan, entitled "Fault-Adaptivity in Hard Real-Time Component-Based Software Systems," develops an approach and model-based support tools to implement software health management functions for component-based systems, where developers can create models of the system and its components, as well as specify how fault mitigation will take place. The foundation of the architecture is a real-time component framework that defines a component model for the ARINC-653 system.

Sousa's paper, entitled "Towards User Tailoring of Self-adaptation in Ubiquitous Computing," argues that domain experts and end users will play an increasingly important role in designing ubiquitous computing systems. He proposes language constructs for users to express and tailor the following kinds of self-adaptation: design meshing—dynamic adaptation to requirements independently put forth by multiple users, pliable applications—structural modes of operation in response to context or other events, and a decentralized, lightweight protocol for self-healing.

The third paper, entitled "Hierarchical Self-Optimization of SaaS Applications in Clouds," by Simmons, Ghanbari, Liaskos, Litoiu, and Iszlai, defines a framework to manage a software as a service (SaaS) application on top of a platform as a service (PaaS) infrastructure. This framework utilizes PaaS policy sets to implement the SaaS provider's elasticity policy for its application server tier. Adaptation is based on strategy-trees to allow for systematic capture, representation, and reasoning about adaptation variability.

The final paper of this part, entitled "Self-Adaptivity from Different Application Perspectives: Requirements, Realizations, Reflections," by Geihs is a reflection on the design space of self-adaptive systems and a critique of a recently published framework for evaluating self-adaptive software systems. This evaluation is performed by elaborating on synergies and discrepancies in the development of three case studies.

Although the self-adaptability of systems has been studied in a wide range of disciplines, from biology to robotics, only recently has the software engineering community 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. Continuing the course of the first book of the series on *Software Engineering for Self-Adaptive Systems* the collection of papers in this second volume addresses the state of the art of the field, describes a wide range of approaches coming from different strands of software engineering, and posits future challenges facing this field of research. We hope that this book will prove valuable for both practitioners and researchers involved in the development and deployment of self-adaptive software systems.

We would like to thank all the authors of the book chapters for their excellent contributions, the participants of the Dagstuhl Seminar 10431 on "Software Engineering for Self-Adaptive Systems" 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: R. Abreu, J. Andersson, L. Baresi, N. Bencomo, Y. Brun, R. Casallas, L. Duchien, F. Eliassen, G. Engels, J. Georgas, V. Grassi, S. Guinea, S. Hallsteinsen, P. Inverardi, S. Jiang, G. Karsai, S. Liaskos, M. Litoiu, A. Lopes, S. Malek, R. Mirandola, J. Mylopoulos, O. Nierstrasz, C. Prehofer, W. Robinson, L. Rodrigues, R. Rouvoy, M. Salehie, B. Schmerl, B. Simons, D. Smith, J.P. Sousa, L. Tahvildari, M. Tichy, D. Weyns, J. Wuttke, and several anonymous reviewers.

June 2012

Rogério de Lemos
Holger Giese
Hausi A. Müller
Mary Shaw

Table of Contents

Part I: Research Roadmap

Part II: Requirements and Policies

Part III: Design Issues

Part IV: Applications

Software Engineering for Self-Adaptive Systems: A Second Research Roadmap

Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw,
Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura,
Norha M. Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi,
Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais,
Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Göschka,
Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai,
Jeff Kramer, Antónia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii,
Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè,
Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Dennis B. Smith,
João Pedro Sousa, Ladan Tahvildari, Kenny Wong, and Jochen Wuttke

r.delemos@kent.ac.uk, holger.giese@hpi.uni-potsdam.de, hausi@cs.uvic.ca,
mary.shaw@cs.cmu.edu

Abstract. The goal of this roadmap paper is to summarize the state-of-the-art and identify research challenges when developing, deploying and managing self-adaptive software systems. Instead of dealing with a wide range of topics associated with the field, we focus on four essential topics of self-adaptation: design space for self-adaptive solutions, software engineering processes for self-adaptive systems, from centralized to decentralized control, and practical run-time verification & validation for self-adaptive systems. For each topic, we present an overview, suggest future directions, and focus on selected challenges. This paper complements and extends a previous roadmap on software engineering for self-adaptive systems published in 2009 covering a different set of topics, and reflecting in part on the previous paper. This roadmap is one of the many results of the Dagstuhl Seminar 10431 on *Software Engineering for Self-Adaptive Systems,* which took place in October 2010.

1 Introduction

The complexity of current software systems has led the software engineering community to investigate innovative ways of developing, deploying, managing and evolving software-intensive systems and services. In addition to the ever increasing complexity, software systems must become more versatile, flexible, resilient, dependable, energy-efficient, recoverable, customizable, configurable, and self-optimizing by adapting to changes that may occur in their operational contexts, environments and system requirements. Therefore, *self-adaptation* — systems that are able to modify their behavior and/or structure in response to their perception of the environment and the system itself, and their goals — has become an important research topic in many diverse application areas.

R. de Lemos et al. (Eds.): Self-Adaptive Systems, LNCS 7475, pp. 1–32, 2013.

It is important to emphasize that in all the many initiatives to explore self-adaption, the common element that enables its provision is usually software. Although control theory provides 60 years of experience and software the necessary flexibility to attain self-adaptability, the proper engineering and realization of self-adaptation in software still remains a formidable intellectual challenge. Moreover, only recently have the first attempts been made to establish suitable software engineering approaches for the provision of self-adaptation. In the long run, we need to establish the foundations that enable the systematic development, deployment, management and evolution of future generations of self-adaptive software systems.

The goal of this roadmap paper is to summarize the state-of-the-art and identify research challenges when developing, deploying, managing and evolving self-adaptive software systems. Specifically, we focus on development methods, techniques, and tools that we believe are required when dealing with software-intensive systems that are self-adaptive in their nature. In contrast to merely speculative and conjectural visions and ad hoc approaches for systems supporting self-adaptability, the objective of this paper is to establish a roadmap for research and identify the key research challenges.

The intent of this new roadmap paper is not to supersede the previous paper on software engineering self-adaptive systems [15], but rather to complement and extend it with additional topics and challenges. The research challenges identified in the previous paper are still valid. Moreover, it is too early to reassess the conjectures made in that paper. In order to provide a context for this roadmap, in the following, we summarize the most important challenges identified in the first roadmap paper [15].

- *Modeling dimensions* — the challenge was to define models that can represent a wide range of system properties. The more precise the models are, the more effective they should be in supporting run-time analyses and decision processes.
- *Requirements* — the challenge was to define a new language capable of capturing uncertainty at an abstract level. Once we consider uncertainty at the requirements stage, we must also find means of managing it. Thus, the need to represent the trade-offs between the flexibility provided by the uncertainty and the assurances required by the application.
- *Engineering* — the challenge was to make the role of feedback control loops more explicit. In other words, feedback control loops must become first-class entities throughout the lifecycle of self-adaptive systems. Explicit modeling of feedback loops will ease reifying system properties to allow their query and modification at run-time.
- *Assurances* — the challenge was how to supplement traditional V&V methods applied at requirements and design stages of development with run-time assurances. Since system context changes dynamically at run-time, systems must manage contexts effectively, and its models must include uncertainty.

Similar to previous research roadmap paper, instead of dealing with a wide range of topics associated with the field, this paper focuses on four essential topics of self-adaptation: design space of self-adaptive solutions, software engineering processes for self-adaptive systems, from centralized to decentralized control, and practical run-time verification and validation (V&V) for self-adaptive systems. The presentations of each of the topics do not cover all related aspects, instead focused theses are used as a means to identify challenges associated with each topic. The four identified theses are the following.

– *Design space* – the need to define what is the design space for self-adaptive software systems, including the decisions the developer should address.
– *Processes* – the need to define innovative generic processes for the development, deployment, operation, maintenance, and evolution of self-adaptive software systems.
– *Decentralization of control loops* – the need to define a systematic engineering approach for control loops for software adaptation of varying degree of centralization and decentralization of the loop elements.
– *Practical run-time verification and validation* – the need to investigate V&V methods and techniques for obtaining inferential and incremental assessments for the provision of confidence and certifiable trust in self-adaptation.

Although the topics covered by the two roadmap papers may appear related, the issues covered are quite distinct since the topics covered different theses: design spaces is related to the former modeling dimensions topic but taking a broader more top-down rather than bottom up perspective, processes is a completely new topic, decentralization of control loops looks into the control loop addressed by the former engineering topic with the particular focus on decentralization, and practical run-time V&V refines the related former assurances topic looking in particular into techniques that can be effectively applied at run-time.

In order to motivate and present a new set of research challenges associated with the engineering of self-adaptive software systems, the paper is divided into four parts, each related to one of the new topics identified for this research roadmap. For each topic, we present an overview, suggest future directions, and focus on selected challenges. The four topics are: design space for adaptive solutions (Section 2), towards software engineering processes for self-adaptive systems (Section 3), from centralized to decentralized control (Section 4), and practical run-time verification and validation (Section 5). Finally, Section 6 summarizes our findings.

2 Design Space

Designing self-adaptive software systems involves making design decisions about observing the environment and the system itself, selecting adaptation mechanisms, and enacting those mechanisms. While most research on self-adaptive systems deals with some subset of these decisions, to the best of our knowledge, there has been neither a systematic study of the overall design space for such systems nor an enumeration of the decisions the developer should address.

2.1 Design Space Definitions

The *design space* of a system is the set of decisions, together with the possible choices, the developer must make. A *representation of a design space* is a static textual or graphical form of a design space, or a subset of that space. Intuitively, a design space is a Cartesian space with dimensions representing the design decisions and values along those dimensions representing the possible choices. Points in the space represent concrete designs. In practice, most interesting design spaces are too rich to represent in their entirety, so representations of the design space capture only the principal decisions, the ones that are significant for the task at hand. Typically, the design dimensions are not independent, so making one decision may preclude, restrict, or make irrelevant, other decisions [9,49].

Several partial methodologies for identifying and representing design spaces have emerged. For example, Andersson *et al.* [1] defined a set of modeling dimensions for self-adaptive software systems. The identified dimensions were organized into four categories: the self-adaptive goals of the system, the causes or triggers of self-adaptation, the mechanisms used to adapt, and the effects of those mechanisms on the system. Kramer and Magee [33] outline three tiers of decisions the developer must make — ones that pertain to goal management, change management, and component control. Dobson *et al.* [17] identify four aspects of self-adaptive systems around which decisions can be organized: collect, analyze, decide, act. Finally, Brun *et al.* [10] discuss the importance of making the adaptation control loops explicit during the development process and outline several types of control loops that can lead to adaptation. Specific design spaces have also been proposed in the form of taxonomies. For example, Brake *et al.* [8] introduce (and Ghanbari *et al.* [22] later refine) a taxonomy for performance monitoring of self-adaptive systems together with a method for discovering parameters in source code. Ionescu *et al.* [29] formally define controllability and observability for web services and show that controllability can be preserved in composition.

2.2 Key Design Space Dimensions

In this section, we outline a design space for self-adaptive systems with five principal dimensions — clusters of design decisions pertinent to self-adaptive systems. The clusters are: observation, representation, control, identification, and enacting adaptation. Each cluster provides additional structure in the form of questions a developer should consider when designing such a system. The *Observation* cluster answers questions related to what is monitored by the self-adaptive system, when and how often monitoring occurs, and how states are determined based on the monitored data. The *Representation* cluster is concerned with the run-time representation of adaptation targets, inputs, effects, system indicators, and states. The *Control* cluster is concerned with the mechanisms through which a solution is enacted. The *Identification* cluster is concerned with the possible solution instances that can achieve the adaptation targets. Finally, the cluster of *Enacting Adaptation* concerns how adaptation can be achieved. While we hope our enumeration will help formalize and advance the understanding for self-adaptive system design, it is not intended to be complete and further work on expanding and refining this design space is necessary and appropriate.

To explain the concepts, we separate self-adaptive systems into two elements: the Managed System, which is responsible for the system's main functionality, and the Adaptation System, which is responsible for altering the Managed System as appropriate. The elements inherent to the managed system (that is, the things that would exist even if it were not adaptively managed) such as the inputs and the environment are captured and used by the Adaptation System. The Adaptation System produces adaptations that impact the Managed System.

Observation. The observation cluster is concerned with design decisions regarding what information is observed by the self-adaptive system and when such observations are made.

A key design decision about self-adaptive systems is "what information will the system observe?" In particular, "what information about the external environment and about the system itself will need to be measured or estimated?" To make these measurements, the system will need a set of sensors; these determine what the system *can* observe. Some of the measurements can be made implicitly (e.g., by inferring them from the state of the system or success or failure of an action). Choices include different aspects of goals, domain knowledge, environment, and the system itself necessary to make decisions about adaptation toward meeting the adaptation goals.

Given the set of information the system observes, another important design decision is "how will the system determine that information?" The system could make direct measurements with sensors, infer information from a proxy, extrapolate based on earlier measurements, aggregate knowledge, etc.

Given a way to observe, there are two important decisions that relate to timing: "what triggers observation?" and "what triggers adaptation?" The system could be continuously observing or observation could be triggered by an external event, a timer, an inference from a previous observation, deviation or error from expected behavior, etc. Thus, the observation can happen at a fixed delay, on-demand, or employ a best-effort strategy. The same decisions relate to when the adaptation triggers, which is also relevant to the control cluster.

Handling uncertainty in the measurements is another decision related to observation. Filtering, smoothing, and redundancy are just some of the solutions to dealing with noise and uncertainty.

Representation. The representation cluster is concerned with design decisions regarding how the problem and system are represented at run-time. Uncertainty is intrinsic to many self-adaptive systems, so some information may be available only at run-time. Therefore, to enable mechanisms of adaptation, key information about the problem and system may have to be made available at run-time. This cluster is closely related to the observation cluster, and has ties to the remaining clusters that make use of this representation.

A key decision in this cluster is "what information is made available to the components of the self-adaptive system?" Answers include different aspects of the adaptation targets, existing domain knowledge, and observations of the environment and Managed System that are found necessary and sufficient for a self-adaptive system to operate.

Another design decision in this cluster relates to choices of internal representation. "How is this information represented?" is meant to guide the designer to the representation that best matches the adaptation targets and the nature of the problem. Choices include explicit representations such as graph models, formulae, bounds, objective functions, etc., or implicit representations in the code.

Control. The control cluster is concerned with the system's run-time decision making toward self-adaptation.

"How to compute how much change to enact forms one design decision in this cluster?" Possible choices include the change being a predefined constant value or proportional to the deviation from the desired behavior. The PID technique adds three values to determine the amount of change: a value proportional to the control error, a value proportional to the derivative of the error, and a value proportional to the integral of the error.

Feedback loops play an integral role in adaptation decisions. Thus, key decisions about a self-adaptive system's control are: "what control loops are involved?" and "how do those control loops interact?" The choices depend on the structure of the system and the complexity of the adaptation goals. Control loops can be composed in series, parallel, multi-level (hierarchical), nested, or independent patterns. Brun *et al.* [10] have further discussed the choices and impact of control loops on the design of self-adaptive systems.

What aspects of the system can be adapted from another design decision? Systems can change parameters, representations, and resource allocations, choose among pre-constructed components and connectors, synthesize new components and connectors, and augment the system with new sensors and actuators.

The possible adaptations those aspects can undergo form another design decision. Choices include aborting, modifying data, calling procedures, starting new processes, etc.

The design decision from the observation cluster that deals with what triggers adaptation is closely related to the control cluster.

Identification. At every moment in time, the self-adaptive system is in one instantiation. The self-adaptation process consists of traversing the space of such instantiations. The identification cluster is concerned with identifying those instantiations the system may take on at run-time. Instantiations can describe system structure, behavior, or both.

For each adaptation target, there is a decision about which instantiations could satisfy that target. The main concern of this decision is enumerating concrete sets of possible structures, behaviors, states, parameter values, etc. It is likely that not all identified instantiations will be supported at run-time. Selecting those that will be supported is another design decision.

Identifying the relevant domain assumptions and contexts for each instantiation is yet another design decision in this cluster. The system can then recognize the context and enact the relevant instantiations.

Finally, identifying the transition cost between instantiations informs the system of the run-time costs of certain types of self-adaptation.

Enabling Adaptation. The choice of adaptation mechanisms the self-adaptive system employs, how are they triggered and supported and how failure is handled are design decisions included in the Enabling Adaptation cluster.

The mechanisms can be represented explicitly or implicitly in the system. For example, self-managing systems with autonomic components typically have explicit adaptation mechanisms. Meanwhile, self-organizing systems often exhibit self-adaptation as an emergent property and do not explicitly define the adaptation mechanisms. The decision concerning control loops from the control cluster is closely related to this decision. Some forms of control can be explicitly expressed in the design, whereas other forms are emergent. It is also possible to create hybrid explicit-implicit self-adaptive systems.

Support of the self-adaptation forms another design decision. Support can be enacted through plugin architectures, component substitution, web services, etc. Related to this decision is what to do when adaptation fails. Choices include trying again, trying a different adaptation mechanism or strategy, observing the environment and the system itself to update internal representations, etc.

In selecting the adaptation mechanisms, it is important to consider the states or events that can trigger adaptation. Examples triggers include not satisfying the adaptation targets that relate to non-functional requirements (e.g., response time, throughput, energy consumption, reliability, fault tolerance), behavior, undesirable events, state maintenance, anticipated change, and unanticipated change.

2.3 Research Challenges

The design space described above can help formalize and advance the understanding of self-adaptive system design. However, it is incomplete, so further exploration and expansion are necessary to aid self-adaptive system developers. A more complete list can help ensure designers avoid leaving out critical decisions.

The main benefit of understanding the design space is infusing a systematic understanding of the options for self-adaptive control into the design process. The developer should understand the trade-offs among different options and the quantitative and qualitative implications of the design choices. To do this effectively, we need to understand the effects of these design decisions, and their order, on the quality of the resulting system.

Each cluster we outlined above can be further expanded and refined. Further, validation of the alternatives against real-world examples can serve as the framework for describing options. Dimensions in the self-adaptive design space are not independent and the interactions between the decisions in those clusters need to be explored. Understanding the decision relationships can narrow the search space and reduce the complexity of the design and the design process.

An important challenge to consider is bridging the gap between the design and the implementation of self-adaptive systems. Frameworks and middleware (e.g., [18,38]) can help bridge that gap, providing developers with automatically generated code and reusable models for their specific design decisions. This challenge is even more difficult in the case of reengineering existing non-self-adaptive systems or integrating self-adaptive and non-self-adaptive systems.

Finally, of particular importance is the understanding of interactions of control loops and self-adaptation mechanisms. If we are to build complex systems, and systems-of-systems with self-adaptive components, we must understand how these mechanisms and their relevant control loops interact and affect one another.

3 Processes

Software engineering (SE) research has primarily focused on principles for developing high quality software systems, while the maintenance and the evolution of such systems have received less attention [40]. Meanwhile, it has been commonly accepted that software, which implements real world applications, must continually evolve. If software does not evolve, it will not fulfill the continuously changing requirements and thus, it will become outdated earlier than expected [35,36]. This awareness has impacted software process models to better address the inherent need for change and evolution by introducing iterative, incremental, and evolutionary approaches to software development as an alternative to strictly separating sequenced disciplines of requirements engineering, design, implementation, and testing [34,40].

In the last decade, software maintenance and evolution have emerged as a key research field in SE [40] that separates the time before and the time after the software is delivered, or in other words, divides the software lifecycle into development-time, deployment-time, and run-time. Post-delivery changes are typically realized by re-entering the regular development disciplines, which eventually results in a new version of a software product or a patch that is then released to replace or enhance the currently running version [32]. Such releases are usually performed during scheduled down-times of the system compromising the system's availability. Thus, the whole maintenance process is mainly performed off-line guided by human-driven change management activities and decoupled from the running system.

However, such a lifecycle does not meet the requirements of *self-adaptive software* [15] that we are considering in this work. A self-adaptive software system operates in a highly dynamic world and must adjust its behavior automatically in response to changes in its environment, in its goals, or in the system itself. This vision requires shifting the human role from operational to strategic. Humans define adaptations, that is, how a system should react to changes and the system then performs adaptations autonomously at run-time. The implication is that activities previously performed during development-time will be shifted to run-time. Several researchers [4,6,27,28] argue that, as a consequence, we have to reconceptualize SE processes to fit modern software systems better. In particular, to fit self-adaptive software systems.

The problem being addressed concerns the timing of software activities [12] in a particular process regarding the software lifecycle. This problem has three dimensions:

- *Software lifecycle phases* [43] (i.e., development, deployment, operation, and maintenance and evolution).
- *Software engineering disciplines* [43] (i.e., requirements, design, implementation, validation, verification, etc.), and activities included in the disciplines (e.g., requirements elicitation, prioritization, and validation).
- *Software activities timeline* [12], that is, when activities take place (i.e., development-time, deployment-time, run-time).

The motivation for our work is that lifecycle activities in a self-adaptive software system are not bound to a traditional timeline (e.g., development-time), but may be shifted to run-time. However, such shifts have uncharted consequences, for instance, they may introduce new process requirements in a different phase (e.g., that additional activities have to be performed during development or deployment of the system to enable shifts of other activities to run-time). Moreover, these consequences must be identified, analyzed, and possibly mitigated, thus resulting in a more dynamic view on software processes. One example of changed timing for activities in self-adaptive systems is verification and validation. The dynamic nature of a running self-adaptive system and its environment requires continuous verification and validation (V&V) to assess the system at run-time. V&V are traditionally performed at development-time and shifting it to run-time requires new and efficient techniques (cf. Section 5). The consequence is a different and more dynamic SE process for self-adaptive systems that needs to be understood and elaborated. Our main challenge is to provide means for engineering processes for self-adaptive systems that will cover the complete software lifecycle. Engineering processes implies support for reasoning about costs and benefits of shifting activities in a process, a prerequisite for engineers to make informed decisions.

3.1 Example: Migrating Evolution Activities

To illustrate the specifics of SE processes for self-adaptive software systems and their differences to traditional software development and evolution activities, we compare the traditional approach to corrective maintenance [50] with the *automatic workarounds* approach [13,14]. Automatic workarounds aim to mask functional faults at run-time by automatically looking for and executing alternative ways to perform actions that lead to failures.

Besides the implementation of new or changing requirements, the evolution of software systems may include corrective maintenance activities [50]. Traditionally, users experience failures and report them to developers who are then in charge of analyzing the failure report, identifying the root cause of the problem, implementing the changes, and releasing the new fixed version of the software.

In contrast, the automatic workarounds mechanism exploits the intrinsic redundancy of "equivalent operations" usually offered by software systems for different needs, but for obtaining the same functionality. Consider for example a container component that implements an operation to *add a single element*, and another operation to *add several elements*. To add two elements, it is possible to add one element after another, or as an equivalent alternative to add them both at the same time using the other operation. If adding two elements in sequence causes a failure at run-time, the automatic workarounds mechanism tries to execute the equivalent operation instead, as an attempt to avoid the problem.

Thus, the automatic workarounds approach partially moves corrective maintenance activities to run-time. Once the user reports a failure, the automatic workarounds mechanism tries to find a workaround based on that information. It checks whether the failure has been experienced by other users and a workaround is already known. If so, it first attempts to execute the workaround known to be valid. If no workaround is known or the known workaround no longer works, the mechanism scans the list of equivalent operations and checks whether they may serve as workarounds.

The automatic workaround mechanism exemplifies how activities, previously performed decoupled from running system instances by software developers similar to development activities, are now performed at run-time by a managing subsystem in a self-adaptive software system. Another example is the *failure analysis* activity, where failure causes are analyzed. In traditional maintenance, the failure report is analyzed by developers while in a self-adaptive software system, the managing subsystem analyzes the failure to find alternative workarounds. In general, compared to traditional SE processes, adding a managing subsystem affects how activities in lifecycle phases are defined and connected. The automatic workarounds approach exemplifies three effects:

- *Migrating activities from one phase to another* — the analyzing failure reports activity is (partially) moved from development-time (maintenance) phase to run-time. This (partially) delegates the developer's responsibility for this activity to a self-adaptation mechanism in the self-adaptive system and it is an example of the effects on the activities' timeline.
- *Introducing new activities in other lifecycle phases* — introducing the automatic workaround mechanism requires that additional activities are performed in the development and maintenance phases. One example is the identification of equivalent operations. Whenever some behavior is "assigned" to the automatic workaround mechanism, equivalent operations for this behavior must be identified.
- *Defining new lifecycle phase inter-dependencies* — the automatic workaround mechanism searches for equivalent operations, executes them, and lets the users evaluate the results. This is repeated either until the user approves the results, and thus the workaround, or until no more equivalent operations could be found. If this is the case, the mechanism is not able to provide a

solution to the problem. The only fallback available is to generate a failure report and send it to the maintenance organization where it will be dealt within the traditional maintenance activity. This exemplifies how traditional maintenance activities integrate with run-time activities, for instance, as information providers or as fallback activities if run-time activities do not succeed.

3.2 Understanding a Self-Adaptive Software System's Lifecycle

Understanding how software is best developed, operated, and maintained is a pervasive research challenge in the SE field. During the last two decades we have witnessed the development of ultra-large-scale, integrated, embedded, and context-aware software systems that have introduced new challenges concerned with system development, operation, and maintenance. For instance, dynamic environments may change systems' goals, the systems' inherent complexity makes it difficult for external parties to be responsible for the operation, and finally, the vast number of systems makes the operations task too complex for a single centralized machine or a system operator. One answer to these advances is to instrument software systems with *managing systems* that make them more autonomous. This autonomy means that systems take over some of the responsibilities previously performed by other roles in the software lifecycle, such as sensing failures and automatically recovering from them.

An SE process is a workflow of activities that are performed by roles and that use and produce artifacts in order to develop, operate, and evolve a software system. In general, we conceive two extreme poles of SE processes [27,28]. One pole corresponds to a traditional, off-line lifecycle process where the system itself has no on-line process activities, that is, no activities are performed by the running system. In contrast, the other pole describes a process with almost all activities performed on-line by the system at run-time. The distinction between off-line and on-line process activities is pivotal for the design of self-adaptive software systems as it enables engineers to design more sophisticated self-adaptation capabilities. In practice, a process for a self-adaptive software system is positioned in between these two extreme poles, due to cost and benefits trade-offs.

The research we envision has as its goal a generic process engineering framework for self-adaptive software systems that provides reasoning support based on the relative costs and benefits for individual design decisions. The framework should include a library of reusable process elements (i.e., activity, role, and artifact definitions). With its built-in support for reasoning, the framework associates value (costs and benefits) with these process elements. These will guide and support engineers in understanding, specifying, analyzing, tuning, and enacting an SE process for a concrete self-adaptive system. The framework is based on process modeling, where models specify processes. Such process descriptions materialize how a self-adaptive software system is developed and how its managing system behaves at run-time. By reifying a process in models, a framework will promote discussions about the process and its design. In the long term, it will promote reuse and even automated analysis of processes [45], which will further support a better understanding of a self-adaptive system's lifecycle.

The key research challenge is the design of the process engineering framework for self-adaptive software systems, which includes three corner-stone components: (i) A library containing definitions of reusable process elements; (ii) Support for specification of concrete self-adaptive software systems' processes; and (iii) Support for reasoning, analysis, and tuning of such process specifications based on their relative costs and benefits.

Definitions of process elements for the library as well as process specifications should be based on an existing framework, such as the *Software & Systems Process Engineering Metamodel Specification* (SPEM) [43]. SPEM provides a modeling language for process specifications including, among others, lifecycle phases, milestones, roles, activities, and work products. Research needs to identify required extensions to SPEM in order to model specifics of processes for self-adaptive systems, like the phases when process elements are employed and their inter-relationships to elements in other lifecycle phases. For example, in the automatic workarounds approach, we identified the analyze failure report activity as one activity that may be performed as part of a regular maintenance phase or at run-time. Another example is to model dependencies between phases (e.g., an activity can only be performed at run-time if another activity has been performed at development-time). In addition, research on how to integrate notions of value, that is, costs and benefits, into SPEM concepts is key. Extensions to SPEM will provide a language for the process engineering framework. A language to define process elements for the library, to model concrete processes for self-adaptive software systems, and to analyze and tune these processes.

The first framework component, defining reusable elements for the generic library, requires a solid understanding of SE processes, self-adaptive systems, and the influential factors such as benefits and costs to a self-adaptive system. This understanding is materialized by those elements that define processes, activities, roles, or artifacts, and is persisted and shared as knowledge, such as best-practices, in the library. Thus, the library supports the understanding and specification of concrete processes by reusing the library's knowledge and element definitions, which is addressed by the second component.

Starting with an abstract conceptual model of the self-adaptive software to be developed and the goals and the environments of the system, an engineer instantiates the library to create a process model for the specific product. The process engineering framework provides methods for decision support and product/process analysis that will assist in the instantiation task. Self-adaptive behavior introduces a complicated bi-directional dependency relation between process modeling and software design. The framework's methods will have to take several factors into consideration including the type of adaptation required at run-time, the associated costs and benefits, and the consequences for other lifecycle activities. In our example, there is a design decision (to use the automatic workaround mechanism) that introduces additional activities as part of the development activities (defining the scope of the mechanism, i.e., which operations should be covered by the mechanism, and identifying equivalent operations for this defined scope).

The third framework component explicitly addresses the product/process analysis and tuning to obtain an enactable process specification that appropriately fits the specific product and the product's goals and environments. A typical sensitivity point is the degree of adaptation and evolution support at run-time. Any design decision concerned with self-adaptive behavior must analyze, for instance, the overhead it introduces. Is the overhead acceptable or not? If not, are precomputed adaptations possible to tune the process by reducing the overhead? As stated in [4], run-time validation and verification may not match the requirements of efficiency to allow the systems to timely react according to changes. This exemplifies that software design and process analysis/tuning are not isolated activities, and it promotes the continuous integration of design decisions and process analysis/tuning throughout a self-adaptive software system's lifecycle.

Finally, it is likely that an engineer uses the three components of the process engineering framework iteratively and concurrently rather than sequentially. For example, while specifying a process, an engineer does not find a suitable process element definition in the library, and thus, new definitions will be created and possibly added to the library. Or during product/process analysis, an engineer identifies the need for process optimization, and searches the library for more suitable process element definitions that could be used to tune the process. Like software development processes, the process of using the framework itself is characterized by incremental, iterative, and evolutionary principles.

Another dimension that should be considered from the beginning is the degree of automation. An absolute requirement is that the process is based on and uses models throughout the lifecycle. Since the system evolves at run-time, these models may also have to evolve (model evolution) and thus, models need to be accessible at run-time, either on-line or off-line [6]. The availability of run-time models makes it possible to use them as interfaces for monitoring [57] and adapting [56] a running system, and to perform what-if analyses and consultations [6] (e.g., to test adaptations at the level of models before actually adapting the running system). In addition, process activities must be based on up-to-date models. Changes in a run-time model allow to some extent for the dynamic derivation of new capabilities to respond to situations unforeseen in the original design. Not all need to be new, we envisage the use of a library of model transformation strategies [4] to derive system models as well as keeping the process up-to-date with respect to the running system and vice versa. As an initial step, model synchronization techniques have already been applied at run-time to keep multiple system models providing different views on a running system up-to-date and consistent to each other [56,57].

3.3 Research Challenges

The different problems and dimensions highlighted in the previous sections can be summarized as research challenges in process comprehension, process modeling, and process design.

First, dynamic environments change the system's goals. As a consequence we need proper means to fully comprehend the characteristics of self-adaptive software systems and the key characteristics of their lifecycles to enhance design & modeling, optimization, and enactment of such systems and processes. For example, more autonomy calls for the capability of self-reacting to anomalous situations. Both probing and reacting must be properly planned, designed, and implemented, and they also require that some activities, which were traditionally performed before releasing the system, be shifted to run-time.

To fully comprehend how software processes change when developing a self-adaptive system also requires that influential factors are identified and understood. Identification of these factors is essential. Factors are costs and benefits related to self-adaptation capabilities. Less complex capabilities may be supported even in a primarily off-line process while more advanced, complex self-adaptation capabilities call for processes where a majority of the activities are performed on-line.

These two challenges impose a proper formalization of the software processes to allow involved parties to fully understand the roles, activities, and artifacts at each stage, but also to increase knowledge and foster reuse. Since some solutions for process definition already exists and SPEM is imposing as one of the most interesting/promising solutions, one should analyze it to understand what can be defined through the standard model, and identify required extensions of this model to take the specifics of processes for self-adaptive systems into account.

Another challenge associated with processes for self-adaptive software systems is the fact that processes need to be generated dynamically at run-time since changes affecting the system, its context and goals may require processes to adapt. However, to deal effectively with the variability associated with software adaptation, it is also necessary to adapt the processes that actually manage the dynamic generation of processes for handling the uncertainty of the changes. This calls for the need to have reflective processes in which a process is adapted by reflecting on itself and its goals. Since off-line and on-line activities might influence each other, another challenge that is identified is the need to consider how the initial development-time design rationale can affect the processes being generated at run-time. The reverse is also crucial, there is the need to incorporate into off-line activities the decisions being made during run-time since they would provide insightful knowledge about the operational profile of the system.

A SPEM-like solution is the enabler for defining a suitable library of generic, reusable process elements. The availability of these elements would turn the definition of suitable software processes, for the different self-adaptive systems, into the assembly of pre-existing blocks with a significant gain in terms of quality, speed, and accuracy of delivered solutions. Orthogonally, it would also allow for the analysis and tuning of designed processes to obtain enactable solutions that appropriately fit different products given their specific stakeholders' goals and environments in which they operate. Accurate analysis and optimization capabilities are mandatory to oversee the actual release of these processes, but they are also important to govern evolution since it is foreseeable that these processes must evolve in parallel with developed systems. Processes must remain aligned and consistent with the corresponding systems and with the environments in which these systems operate. Adequate design support for self-adaptive systems and their lifecycle processes, where value and trade-offs are central, is a remaining grand challenge for engineering self-adaptive software systems.

4 Decentralization of Control Loops

Control loops have been identified as crucial elements to realize the adaptation of software systems [17,30,48]. As outlined in the the former road map [15], a single centralized control component may realize the adaptation of a software system, or multiple control components may realize the adaptation of a composite of software systems in a decentralized manner. In a decentralized setting, the overall system behavior emerges from the localized decisions and interactions. These two cases of self-adaptive behavior, in the form centralized and decentralized control of adaptation are two extreme poles. In practice, the line between the two is rather blurred, and development may result in a compromise. We illustrate this with a number of examples.

Adaptation control can be realized by a simple sequence of four activities: monitor, analyze, plan, and execute (MAPE). Together, these activities form a feedback control system from control theory [47]. A prominent example of such adaptation control is realized in the Rainbow framework [19]. Hierarchical control schemes allow management or the complexity of adaptation when multiple concerns (self-healing, self-protection, etc.) have to be taken into account. In this setting, higher level adaptation controllers determine the set values for the subordinated controllers. A prominent example of a hierarchical control schema is the IBM architectural blueprint [25]. In a fully decentralized adaptation control schema, relatively independent system components coordinate with one another and adapt the system when needed. An example of this approach is discussed in [21] in which component managers on different nodes automatically configure the system's components according to the overall architectural specification.

These examples show that a variety of control schemas for self-adaptive systems are available. Our interest in this section is twofold: first, we are interested in understanding the drivers to select a particular control schema for adaptation; and second, we are interested in getting better insight in the possible solutions to control adaptation in self-adaptive systems. Both the drivers and solutions are important for software engineers of self-adaptive system to choose the right solution concerning centralized or decentralized control. In the remainder of this section, we report on our findings concerning this endeavor and outline some of the major research questions we see to achieve that a systematic engineering approach for designing centralized or decentralized control schemes for software adaptation.

4.1 Distribution versus Decentralization

Before we elaborate on the problems and possible solutions of different control schemas in self-adaptive systems, we first clarify terminology. In particular, we want to clarify the terms distribution and decentralization, two terms that are often mixed by software engineers in the community of self-adaptive systems, leading to a lot of confusion.

Textbooks on distributed systems (e.g., [51]) typically differentiate between centralized data (in contrast to distributed, partitioned, and replicated data), centralized services (in contrast to distributed, partitioned, and replicated services) and centralized algorithms (in contrast to decentralized algorithms).

Our main focus with respect to decentralization is on the algorithmic aspect. In particular, with *decentralization* we refer to a particular type of control in a self-adaptive software system. With control, we mean the decision making process that results in actions that are executed by the self-adaptive system. In a decentralized system there is no single component that has the complete system state information, and the processes make adaptation decisions based only on local information. In a centralized self-adaptive system on the other hand, decisions regarding the adaptations are made by a single component.

With *distribution*, we refer to the deployment of a software system to the hardware. Our particular focus of distribution here is on the deployment of the managed software system. A distributed software system consists of multiple software components that are deployed on multiple processors that are connected via some kind of network. The opposite of a distributed software system is a system consisting of software that is deployed on a single processor.

From this perspective, control in a self-adaptive software system can be centralized or decentralized, independent of whether the managed software is distributed. In practice, however, when the software is deployed on a single processor, the adaptation control is typically centralized. Similarly, decentralized control often goes hand in hand with distribution of the managed software system.

The existing self-adaptive literature and research, in particular those with a software engineering perspective, have by and large tackled the problem of managing either local or distributed software systems in a centralized fashion (e.g., [19,25,44]). While promising work is emerging in decentralized control of self-adaptive software (e.g., [11,21,39,58,59]), we believe that there is a dearth of practical and effective techniques to build systems in this fashion.

It is important to highlight that the adaptation control schema we consider here (from centralized to decentralized control) is just one dimension of the design space of a distributed self-adaptive system. Other aspects of the design space include the actual distribution of the MAPE components, the distribution of the data and supporting services required to realize adaptation, the mechanisms for communication and coordination, etc.

4.2 Drivers for Selecting a Control Schema for Adaptation

Two key drivers for selecting the right control schema for adaptation in self-adaptive systems are the characteristics of the domain and the requirements of the problem at hand.

Domain Characteristics. Specific properties of the domain may put constraints on the selection of a particular control schema for adaptation. We give a number of example scenarios.

- In open systems, it might be the case that no trustworthy authority exists that can realize central control.
- When all information that is required for realizing adaptations is available at the single node, a centralized control schema may be easy to realize. However, in other settings, it might be very difficult or even unfeasible to get centralized access to all the information that is required to perform an adaptation.
- The communication network may be unreliable causing network disruptions that require decision making for adaptations based on local information only.

Requirements of the Problem at Hand. Stakeholder requirements may exclude particular solutions to realize adaptations.

If optimization is high on the priority list of requirements, a centralized approach may be easier to develop and enables optimization to be rather straightforward. On the other hand, in a decentralized approach, meeting global goals is known to be a complex problem. Hence, we have to compromise on the overall optimality in most cases.

For systems in which guarantees about system wide properties are crucial, fully decentralized solutions can be very problematic. Decentralized control imposes difficult challenges concerning consistency, in particular in distributed settings with unreliable network infrastructures. However, if reaction time is a priority, exchanging all monitored data that is required for an adaptation may be too slow (or too costly) in a centralized setting.

When scalability is a key concern, a decentralized solution may be preferable. Control systems with local information scale well in terms of size, and also regarding performance as the collection of information and control implementation are local. In contrast, scalability in a centralized setting is limited as all control information must be collected and processed at the single control point.

A central control scheme is also less robust as it results in a single point of failure. In a decentralized setting, when subsystems get disconnected, they may be able to operate and make decisions based on the local information only, hence increasing robustness.

4.3 Patterns for Interacting Control Loops

Ideally, we would like to have a list of problem characteristics/requirements and then match solutions against these. However, in practice, as stakeholders typically have multiple, often conflicting requirements, any solution will imply trade-offs.

We have identified different solutions in the form of patterns of interacting control loops in self-adaptive systems. Patterns are an established way to capture design knowledge fostering comprehension of complex systems, and serving as the basis for engineering such systems. Each pattern can be considered as a particular way to orchestrate the control loops of complex self-adaptive software systems, as we explained in Section 2.2.

In order to describe the different patterns, we consider the interactions among the different phases of control loops realized by the MAPE components. Typically only the M and E phases interact with the managed system (to observe and adapt the system respectively). Furthermore, we consider possible peer interactions among phases of any particular type (e.g., interactions between P phases), and interactions among phases that are responsible for subsequent phases (e.g., an A phase interacts with a P phase, or a P phase that interacts with an E phase). According to the different interaction ways we have identified five different patterns that we briefly illustrate in the following.

Pattern 1: Hierarchical Control. In the hierarchical control pattern, the overall system is controlled by a hierarchical control structure where complete MAPE loops are present at all levels of the hierarchy. Generally, different levels operate at different time scales. Lower levels loops operate at a short time scale, to guarantee timely adaptation concerning the part of the system under their direct control. Higher levels operate at a longer time scale and with a more global vision. MAPE loops at different levels interact with each other by exchanging information. The MAPE loop at a given level may pass to the level above information it has collected, possibly filtered or aggregated, together with information about locally planned actions, and may issue to the level below directives about adaptation plans that should be refined into corresponding actions.

This pattern naturally fits systems with a hierarchical architecture. However, independently of the actual system architecture, hierarchical organization of the control system has been proposed (e.g., in [33]) to get a better separation of concerns among different control levels.

Pattern 2: Master/Slave. The master/slave pattern creates a hierarchical relationship between one master that is responsible for the analysis and planning part of the adaptation and multiple slaves that are responsible for monitoring and execution. Figure 1 shows a concrete instance of the pattern with two slaves.

In this case, the monitor components M of the slaves monitor the status of the local managed subsystems and possibly their execution environment and send the relevant information to the analysis component A of the master. A, in turn, examines the collected information and coordinates with the plan component P, when a problem arises that requires an adaptation of the managed system. The plan component then puts together a plan to resolve the problem and coordinates with the execute components (E) on the slaves to execute the actions to the local managed subsystems.

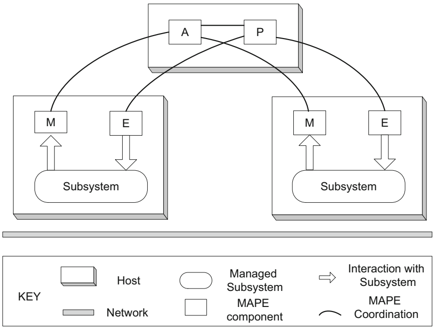


Fig. 1. Master-slave pattern

The master/slave pattern is a suitable solution for application scenarios in which slaves are willing to share the required information to allow centralized decision making. However, sending the collected information to the master node and distributing the adaptation plans may impose a significant communication overhead. Moreover, the solution may be problematic in case of large-scale distributed systems where the master may become a bottleneck.

Pattern 3: Regional Planner. In the regional planner pattern, a (varying) number of local hosts are hierarchically related to a single regional host. The local hosts are responsible for monitoring, analyzing and executing, while the regional host is in charge of the planning part. In this case, the monitor component M of each local host monitors the status of the managed subsystem and possibly its execution environment, and the local analysis component A analyzes the collected information, and reports the analysis results to the associated regional plan component P. P collects this information from all the hosts under its direct supervision, thus acquiring a global knowledge of their status. The regional P is in charge to evaluate the need of adaptation of the managed system and, in case, to elaborate an adaptation plan to resolve the problem, coordinating its decisions with other peer regional plan components. The plan can then be put in action activating the execute components E on the local hosts involved in the adaptation.

Regional planner is a possible solution to the scalability problems with master/slave. Regions may also map to ownership domains where each planner is responsible for the planning of adaptations of its region.

Pattern 4: Fully Decentralized. In this pattern, each host implements a complete MAPE loop, whose local M, A, P and E components coordinate their operation with corresponding peer components of the other hosts. Ideally, this should lead to a flexible sharing of information about the status of the managed systems, as well as the results of the analysis. The triggering of possible adaptation actions is then agreed on and managed by the local P components, which then activate their local E components to execute the actions to the local managed subsystems. In practice, achieving a globally consistent view on the system status, and reaching a global consensus about suitable adaptation actions is not an easy task. In this case, it could be preferable to limit the interaction among peer control components to get some partial information sharing and some kind of lose coordination. Generally, this may lead to sub-optimal adaptation actions, from the overall system viewpoint. However, depending on the system at hand and the corresponding adaptation goals, even local adaptation actions based on partial knowledge of the global system status may lead to achieve globally optimal goals (TCP adaptive control flow is a typical example of this).

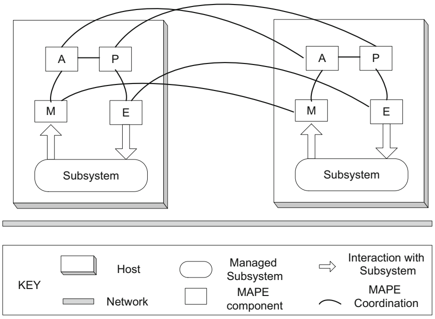


Fig. 2. Decentralized pattern

Pattern 5: Information Sharing. In this pattern, each host owns local M, A, P and E components, but only the monitor components M communicates with the corresponding peer components. Therefore the information collected about the status of the managed systems is shared among the various monitors, while the analysis of the collected data and the decision about possible adaptation actions taken by the plan components P are performed without any coordination action with the other hosts.

Information sharing is for example useful in peer-to-peer systems where peers can perform local adaptations but require some global information. One possible approach to share such global information is by using a gossip approach.

4.4 Outlook

So far, the research community on self-adaptive and autonomic systems has spent little effort in studying the interactions among components of MAPE loops. Our position of making the control loops explicit underlines the need for a disciplined engineering practice in this area. Besides the consolidation of architecture knowledge in the form of different MAPE configurations as patterns, we also need practical interface definitions (signatures and APIs), message formats, and protocols. The necessity of such definitions has partially already been appreciated in the past, e.g., in [37] the authors standardize the communication from the A to the P component by using standard BPEL (Business Process Execution

Language) as the data exchange format, but no comprehensive approach exists so far.

In terms of future research, there are a number of interesting challenges that need to be investigated when considering different self-adaptive control schemes, including:

- *Pattern applicability* — in what circumstances and for what systems are the different patterns of control applicable? Which quality attribute requirements hinder or encourage which patterns? What styles and domains of software are more easily managed with which patterns?
- *Pattern completeness* — what is the complete set of patterns that could be applied to self-management?
- *Quality of service analysis* — for decentralized approaches, what techniques can we use to guarantee system-wide quality goals? What are the coordination schemes that can enable guaranteeing these qualities?

We already mentioned the need for studying other aspects of the design space of adaptation in self-adaptive software systems, including distribution of the MAPE components, distribution of the data and supporting services required to realize adaptation, etc.

Finally, there may be a relationship between the architecture of the managed system and the architecture of the management system. How do we characterize this relationship and help us to choose the appropriate management patterns for the appropriate systems?

5 Practical Run-Time Verification and Validation

In a 2010 science and technology research agenda for the next 20 years, US Air Force (USAF) chief scientist Werner Dahm identified *control science* as a top priority for the USAF [60]. Control science can be defined as a systematic way to study certifiable validation and verification (V&V) methods and tools to allow humans to trust decisions made by self-adaptive systems. According to Dahm, the major barrier preventing the USAF from gaining more capability from autonomous systems is the lack of V&V methods and tools. In other words run-time V&V methods and tools are critical for the success of autonomous, autonomic, smart, self-adaptive and self-managing systems.

While high levels of adaptability and autonomy result in obvious benefits to the stakeholders of software systems, realizing these abilities with confidence is hard. Designing and deploying certifiable V&V methods for self-adaptive systems (SAS) is one of the major research challenges for the software engineering community in general and the self-adaptive systems community in particular. It may take a large part of this decade, if not more, investigating these research challenges to arrive at effective and practical solutions [60].

The V&V roadmap chapter in this book, entitled "Towards Practical Run-time V&V of Self-Adaptive Software Systems," provides a vision of open challenges and discusses run-time V&V challenges from several perspectives:

(i) contrasting design-time and run-time V&V; (ii) defining adaptation properties and viability zone dynamics for SAS; (iii) making V&V explicit in the self-adaptation loops of SAS; (iv) characterizing run-time V&V enablers (i.e., requirements at run-time, models at run-time, and smart context); and (v) ensuring adaptive control.

5.1 Run-Time V&V Research Enablers

Foundational Questions and the Viability Zone. One systematic approach to control science for adaptive systems is to study V&V methods for the mechanisms that sense the dynamic environmental conditions and the target system behavior, and act in response to these conditions by answering the fundamental questions: (i) *what* to validate? (ii) *where* to measure the aspects to validate? and (iii) *when* to validate these aspects? The *what* refers to the system's requirements and adaptation properties that must be validated and verified. The *where* relates to the separation of concerns between the target system and the adaptation mechanism (where V&V must be applied). Finally, the *when* corresponds to the stages of the adaptation process in which V&V tasks are to be performed. The answers to these questions determine the V&V methods that are suitable to keep a particular adaptive system operating within its viability zone. We define the *viability zone* of an adaptive system as the set of possible states in which the system's requirements and desired properties are satisfied [3].

Dependency on Dynamic Context Monitoring. Viability zones are highly dependent on relevant context entities. Relevant context entities provide the attributes to characterize the dimensions of a viability zone.

Viability zones are dynamic. Every time the adaptation process modifies either the target system or the adaptation controller, new variables are added to, or existing ones are replaced by others in the viability zone. Changes in requirements or adaptation goals can affect also the viability zone. Therefore, dynamic context monitoring is an important requirement for run-time V&V tasks, since the coherence of the monitoring infrastructure with respect to the system goals can be compromised. Dynamic context monitoring exploits models and requirements at run-time to maintain an up-to-date and explicit relationship between system requirements and monitoring strategies. This explicit representation and monitoring allow SAS to recognize changes in requirements and then to trigger changes in monitoring strategies accordingly [52,54,55].

5.2 Run-Time V&V Research Directions

Software validation and verification (V&V) concerns the quality assessment of software systems throughout their lifecycle. The goal is to ensure that the software system satisfies its functional requirements and meets its expected quality attributes [7,26]. To establish "certifiable trust" in software systems that adapt themselves according to contextual and environmental changes at run-time, we need powerful and versatile V&V methods, techniques, and tools. A promising research direction is to ease or relax the traditional software engineering approach, where we satisfy requirements outright, to a more control engineering approach, where we regulate the satisfaction of functional and particularly non-functional requirements using feedback loops [41]. To accomplish this, adaptive software assesses its own situation with respect to its goals continuously, and uses different adaptation strategies accordingly. Nevertheless, the system itself must ensure that its desired behavior is not compromised as a result of the adaptation process. This is particularly important for safety-critical applications.

Quality assessment of self-adaptive software involves both the immutable and the adaptive parts of the system. For the immutable parts, traditional V&V techniques are sufficient. However, for the adaptive parts, the engineering of self-adaptive software requires the development of new, or the tailoring of traditional V&V methods to be applied at run-time throughout the adaptation process. The *Models@run-time* and *Requirements@run-time* research communities provide valuable support for validating and monitoring run-time behavior with respect to the system's requirements [6,46]. The term control science is an appropriate term to characterize this research realm that combines self-adaptation with run-time V&V techniques to regulate the satisfaction of system requirements. It is critical for the SEAMS community to develop a control science involving design-time and run-time V&V methods and techniques for self-adaptive and self-managing systems with inferential, incremental and compositional characteristics that provide adequate confidence levels and certifiable trust in the self-adaptation processes of such systems.

An important first step towards practical run-time validation and verification of self-adaptive software systems is to make V&V tasks explicit in the elements of feedback adaptation loops. This means, for example, to add a V&V component to every phase of the MAPE-K loop [30]. V&V enablers (i.e., requirements at run-time, models at run-time, and dynamic context monitoring) provide effective support to materialize V&V assurances for self-adaptation. Models at run-time enable the validation and monitoring of run-time behavior by providing on-line abstractions of the system state and its relevant context [2,5]. Requirements at run-time provide V&V tasks with on-line representations of the system requirements and adaptation properties throughout the adaptation process [46]. Dynamic context monitoring enables run-time V&V with relevant monitoring mechanisms that keep track of aspects to validate, even when monitoring requirements change at run-time [54].

5.3 Research Challenges

We argue that the fundamental problems addressed by run-time V&V for self-adaptive systems are identical to those of traditional, design-time V&V [20]. That is, independent of the self-* adaptation goals, context awareness, and even uncertainty, V&V fundamentally aims at guaranteeing that a system meets its requirements and expected properties. One key differentiating factor between run-time and design-time V&V is that resource constraints such as time and computing power are more critical for run-time V&V. From these constraints, non-trivial challenges arise, and to tackle them we should depart of course from traditional V&V methods and techniques. On the one hand, these formal V&V methods are often too expensive to be executed regularly at run-time when the system adapts due to their time and space complexity. On the other hand, context-dependent variables are unbound at design time, but bound at run-time. Thus, performing V&V on these variables at run-time is valuable to reduce the verification space significantly, even when the SAS system viability zone varies with context changes. From this perspective, it is crucial to determine precisely when in the adaptation process these V&V operations are to be performed to guarantee the system properties and prevent unsafe operation.

V&V Techniques: Desirable Properties. Even though traditional V&V techniques (e.g., testing, model checking, formal verification, static and run-time analysis, and program synthesis) have been used for the assessment of quality attributes such as those mapped to adaptation properties by Villegas *et al.* [53], an important challenge is their integration into the self-adaptation lifecycle (i.e., at run-time). This integration requires yet another kind of properties—properties on V&V techniques—including sensitivity, isolation, incrementality, and composability.

According to González *et al.*, sensitivity and isolation refer to the level of run-time testability that an adaptive software system can support [24]. On the one hand, sensibility defines the degree to which run-time testing operations interfere with the running system services delivery. That is, the degree in which run-time V&V may affect the accomplishment of system requirements and adaptation goals. Examples of factors that can affect run-time test sensitivity are (i) component state, not only because run-time tests are influenced by the actual state of the system, but because the state of the system could be altered as a result of test invocations; (ii) component interactions, as the run-time testability of a component may depend on the testability of the components it interacts with; (iii) resource limitations, because run-time V&V may affect non-functional requirements such as performance at undesirable levels; and (iv) availability, as run-time validation can be performed depending on whether testing tasks require exclusive usage of components with high availability requirements. On the other hand, González *et al.* also define isolation as the means to counteract run-time test sensitivity. Instances of techniques for implementing test isolation are (i) state separation (e.g., blocking the component operation while testing takes place, performing testing on cloned components); (ii) interaction separation (e.g., blocking component interactions that may be propagated due to results of test invocations); (iii) resource monitoring (e.g., indicating that testing must be postponed due to resources unavailability); and (iv) scheduling (e.g., planning testing executions when involved components are less used).

Requirements and Models at Run-Time. Requirements define the objectives of validation and verification for software systems. However, adaptive systems requirements are dynamic and subject to change at run-time. Thus, these systems require suitable V&V techniques to cope with the dynamics after behavioral and structural changes. From this perspective, the application of run-time automatic testing techniques to enable adaptive software systems with self-testing capabilities seems to be a promising approach. An instance of this approach is the self-testing framework for autonomic computing systems proposed by King *et al.* [31]. This framework dynamically validates change requests in requirements using regression testing and customized tests to assess the behavior of the system under the presence of added components. For this, autonomic managers designed for testing are integrated into the current workflow of autonomic managers designed for adaptation. Two strategies support their validation process: (i) safe adaptation with validation, and (ii) replication with validation. In the first strategy, testing autonomic managers apply an appropriate validation policy during the adaptation process where involved managed resources are blocked until the validation is completed. If the change request is accepted, the corresponding managed resources are adapted. In the second strategy, the framework maintains copies of the managed resources for validation purposes. Thus, changes are implemented on copies, then validated, and if they are accepted, the adaptation is performed. Testing policies can also be defined by administrators and loaded into the framework at run-time.

This self-testing approach illustrates the blurred boundaries among the software lifecycle phases and the many implications of V&V for self-adaptive software systems. Some of these implications constitute challenges that arise from requirements engineering, and model-driven engineering. First, run-time V&V tasks rely on on-line representations of the system and its requirements. Second, requirements at run-time support requirements traceability to identify incrementally what to validate, the requirements subset that has changed, and when. Moreover, test case priority further contributes to refine this incremental validation. Third, for context-aware requirements, run-time models must explicitly define the mapping between environmental conditions that must be monitored at run-time, and corresponding varying requirements. Furthermore, models are useful to support the dynamic reconfiguration of monitoring strategies according to changes in requirements and the execution environment. The Requirements@run-time and Models@run-time research communities provide valuable foundations for run-time V&V of self-adaptive software systems [2,5,46].

Context Uncertainty. To cope with context uncertainty, some of the previously proposed approaches to manage unexpected context changes, fully automated or human-assisted, can be exploited. For instance, Murray *et al.* used feedback loops to cover, with respect to the system requirements, the broadest possible range of system states to transition among them by adaptation operations. Their strategy is to augment robustness by reducing context uncertainty [42]. The approach by Goldsby and Cheng uses state machines to model adaptive systems with transitions as system reconfiguration [23]. Inspired by the adaptability of living organisms, they model systems using UML diagrams and apply digital evolution techniques to generate not only one, but several target states for a given transition, and then assist the user to select the one most appropriate. Thus, they address context uncertainty by generating several possible target system states with qualitatively different QoS characteristics, all of them satisfying the required QoS conditions.

On the side of exhaustive V&V methods, model checking has been used at design time to verify desired properties or conditions on software systems to overcome the limitations of testing techniques, based on a correctness specification. The well known practical problem of this method is the state explosion, which implies the representation of all of the states of the system behavior. In self-adaptive software, this problem is augmented given its changing nature. In effect, the software structure of this kind of systems is subject to re-configuration operations (e.g., adding/removing components and their interconnections) in response to context changes at run-time. Thus, in contrast to the checking requirements of structural static configuration of traditional software, in self-adaptive systems model checking must be applied to each of the possible configurations produced by adaptation mechanisms.

The validation and verification of self-adaptive software systems at run-time is an urgent necessity, and a huge challenge to establish "certifiable trust" in practical adaptation mechanisms. However, despite the development of run-time V&V methods is necessary and plays an important role in the quest towards achieving effective run-time V&V, they are insufficient. To reason effectively and provide assurances on the behavior of self-adaptive systems at run-time, a promising approach is to combine control and software engineering techniques. We aptly termed this combination of foundational theories and principles for run-time V&V methods *control science.*

This section discussed important challenges and possible roadblocks for run-time validation and verification of self-adaptive systems. First, the traceability of evolving requirements, and run-time representations of system aspects are crucial for the identification of what to validate and when. Concrete issues concerning the answers to these questions appear when deciding in which phase of the adaptation loop to implement run-time V&V techniques. Second, these techniques must exhibit desirable properties thus increasing their complexity. Third, dynamic instrumentation such as dynamic monitoring is also required to realize run-time V&V techniques to be implemented throughout the adaptation process.

The assessment of research approaches on self-adaptive software systems constitutes an important starting point for the development of standardized and objective certification methods. For this, we believe that the evaluation framework proposed by Villegas *et al.* provides useful guidance and insights [53]. The SEAMS community is ideally positioned to conduct ground-breaking control science research in our quest towards certifiable trust in self-adaptation.

6 Overall Challenges

In this section, we present the overall conclusions of the research roadmap paper in the context of the major ensuing challenges for our community. First and foremost, this exercise was not intended to be exhaustive. We decided to focus on four major topics identified as key to engineering of self-adaptive software systems: design space of self-adaptive solutions, software engineering processes for self-adaptive systems, from centralized to decentralized control, and practical run-time verification and validation (V&V) for self-adaptive systems. We now summarize the most important challenges for each topic.

- *Design space* — a major challenge associated with design space is to infuse a systematic understanding of the alternatives for adaptive control into the design process. Since the alternatives could be represented as clusters of design decisions, another challenge should be the detailed refinement of dimensions that characterize these clusters in order to provide a complete set of choices to the developer. Moreover, since dimensions should not be dependent, the search space for the solution can be reduced by identifying the dependencies between the different dimensions. Another identified challenge is how to map a generalized design space into an implementation.
- *Processes* — there are two key challenges related to software processes for self-adaptive systems, first, to have a full understanding of the nature of system, its goals and lifecycle in order to establish appropriate software processes, and second, to understand how processes changes and what are the factors affecting these changes. Another major challenge is the formalization of processes for understanding the roles, activities, and artifacts at each stage of the process. This formalization would enable the definition of a library of generic and reusable entities that could be used across different self-adaptive software systems, and would also facilitate the analysis and tuning of processes according to the system.
- *Decentralization of control loops* — since the direction taken in this topic was the identification of patterns for capturing the interaction of control loops in self-adaptive systems, most of the challenges identified are associated with patterns. For example, concerning pattern applicability, what are the circumstances that decide the applicability of patterns, and what application domains or architectural styles that are better managed by patterns? Also there is the challenge of identifying a complete set of patterns that could be applied to the management of self-adaptive systems. Outside the context of patterns, when considering decentralized approach, a major challenge would be to identify techniques that can be used for guaranteeing system-wide quality goals, and the coordination schemes that enable guaranteeing these qualities.
- *Practical run-time verification and validation* — three key challenges related to the run-time verification and validation of self-adaptive software systems were identified. The first challenge is associated with the need to trace the evolution of requirements in order to identify what and when to validate, and the V&V method to be employed. The second challenge is to control the inevitable complexity that is expected from run-time V&V techniques, and final challenge is related to the need of providing appropriate dynamic monitoring when employing run-time V&V techniques.

There are several topics related to software engineering for self-adaptive systems that we did not cover, some of which we now mention, and which can be considered key challenges on their own. First, how to design in an integrated way self-adaptive system in order to enable them to handle expected and unexpected changes? For example, when composing systems designs should provide some elasticity in order to improve their robustness when reacting to changes. Another issue related to system design is whether adaptation should be reactive or proactive. Further, how should competition and cooperation be managed? How to integrate development-time and run-time V&V in order to provide the necessary assurances before deploying a self-adaptive system? Still related to run-time V&V, what kind of processes and how these should be deployed in order to manage the collection, structuring and analysis of evidence? One of the key activities of feedback control loops in self-adaptive software systems is decision making, and its associated adaptation techniques and criteria for balancing, for example, quality of services, over-provisioning, and cost of ownership. Underpinning all the above issues is the question what shape should take a comprehensive system theory, or theories, for self-adaptive software systems [16]? We also did not cover technologies like model-driven development, aspect-oriented programming, and software product lines. These technologies might offer new opportunities and approaches in the development of self-adaptive software systems. Finally, we did not discuss exemplars — canonical problems and accompanying self-adaptive solutions — which are a likely stepping stone to the necessary benchmarks, methods, techniques, and tools to solve the challenges of engineering self-adaptive software systems.

The four topics discussed in this paper outline challenges that our community must face in engineering self-adapting software systems. All these challenges result from the dynamic nature of self-adaptation, which brings 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 for developing, deploying, managing and evolving self-adaptive software systems.

References

1. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Modeling Dimensions of Self-Adaptive Software Systems. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 27–47. Springer, Heidelberg (2009)
2. Aßmann, U., Bencomo, N., Cheng, B.H.C., France, R.B.: Models@run.time (Dagstuhl Seminar 11481). Dagstuhl Reports 1(11), 91–123 (2012), `http://drops.dagstuhl.de/opus/volltexte/2012/3379`
3. Aubin, J., Bayen, A., Saint-Pierre, P.: Viability Theory: New Directions. Springer, Heidelberg (2011), `http://books.google.ca/books?id=0YpZNVBXNK8C`
4. Baresi, L., Ghezzi, C.: The disappearing boundary between development-time and run-time. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER 2010), pp. 17–22. ACM, New York (2010), `http://doi.acm.org/10.1145/1882362.1882367`

5. Bencomo, N., Blair, G., France, R., Muñoz, F., Jeanneret, C.: 4th International Workshop on Models@run.time. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 119–123. Springer, Heidelberg (2010)
6. Blair, G., Bencomo, N., France, R.B.: Models@run.time: Guest Editors' Introduction. IEEE Computer 42(10), 22–27 (2009)
7. Bourque, P., Dupuis, R.: Guide to the Software Engineering Body of Knowledge (SWEBOK). IEEE Computer Society (2005), http://www.computer.org/portal/web/swebok/home
8. Brake, N., Cordy, J.R., Dancy, E., Litoiu, M., Popescu, V.: Automating discovery of software tuning parameters. In: Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems, SEAMS 2008, pp. 65–72. ACM, New York (2008), http://doi.acm.org/10.1145/1370018.1370031
9. Brooks, F.P.: The Design of Design: Essays from a Computer Scientist, 1st edn. Addison-Wesley Professional (2010)
10. 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.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009)
11. 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 2007), Minneapolis, MN, USA (May 2007)
12. Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G.: Towards a taxonomy of software change. Journal of Software Maintenance and Evolution: Research and Practice 17(5), 309–332 (2005), http://dx.doi.org/10.1002/smr.319
13. Carzaniga, A., Gorla, A., Perino, N., Pezzè, M.: Automatic workarounds for web applications. In: FSE 2010: Proceedings of the 2010 Foundations of Software Engineering Conference, pp. 237–246. ACM, New York (2010)
14. Carzaniga, A., Gorla, A., Pezzè, M.: Self-healing by means of automatic workarounds. In: SEAMS 2008: Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems, pp. 17–24. ACM, New York (2008)
15. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
16. Dobson, S., Sterritt, R., Nixon, P., Hinchey, M.: Fulfilling the vision of autonomic computing. Computer 43(1), 35–41 (2010)
17. Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A survey of autonomic communications. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 1, 223–259 (2006)
18. Elkhodary, A., Esfahani, N., Malek, S.: FUSION: A framework for engineering self-tuning self-adaptive software systems. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2010), Santa Fe, NM, USA, pp. 7–16 (2010)

19. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer 37, 46–54 (2004)
20. Gat, E.: Autonomy software verification and validation might not be as hard as it seems. In: Proceedings 2004 IEEE Aerospace Conference, pp. 3123–3128 (2004)
21. Georgiadis, I., Magee, J., Kramer, J.: Self-Organising Software Architectures for Distributed Systems. In: 1st Workshop on Self-Healing Systems. ACM, New York (2002)
22. Ghanbari, H., Litoiu, M.: Identifying implicitly declared self-tuning behavior through dynamic analysis. In: International Workshop on Software Engineering for Adaptive and Self-Managing Systems, pp. 48–57 (2009)
23. Goldsby, H.J., Cheng, B.H.C.: Automatically Generating Behavioral Models of Adaptive Systems to Address Uncertainty. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 568–583. Springer, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-87875-9_40
24. González, A., Piel, E., Gross, H.G.: A Model for the Measurement of the Runtime Testability of Component-Based Systems. In: Proceedings of 2009 International Conference on Software Testing Verification and Validation Workshops, pp. 19–28. IEEE (2009), http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4976367
25. IBM: An architectural blueprint for autonomic computing. Tech. rep. IBM (January 2006)
26. IEEE: Industry implementation of international standard ISO/IEC 12207:95, standard for information technology-software life cycle processes. Tech. rep. IEEE (1996)
27. 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), http://dx.doi.org/10.1007/978-3-540-75336-0_5
28. Inverardi, P., Tivoli, M.: The Future of Software: Adaptation and Dependability. In: De Lucia, A., Ferrucci, F. (eds.) ISSSE 2006-2008. LNCS, vol. 5413, pp. 1–31. Springer, Heidelberg (2009), http://www.springerlink.com/content/g624t1466m9v5647/
29. Ionescu, D., Solomon, B., Litoiu, M., Iszlai, G.: Observability and controllability of autonomic computing systems for composed web services. In: 6th IEEE International Symposium on Applied Computational Intelligence and Informatics, SACI 2011 (2011)
30. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
31. King, T.M., Ramirez, A.E., Cruz, R., Clarke, P.J.: An Integrated Self-Testing Framework for Autonomic Computing Systems. Journal of Computers 2(9), 37–49 (2007), http://academypublisher.com/ojs/index.php/jcp/article/view/361
32. Kitchenham, B.A., Travassos, G.H., von Mayrhauser, A., Niessink, F., Schneidewind, N.F., Singer, J., Takada, S., Vehvilainen, R., Yang, H.: Towards an ontology of software maintenance. Journal of Software Maintenance: Research and Practice 11(6), 365–389 (1999), http://dx.doi.org/10.1002/(SICI)1096-908X(199911/12)11:6<365::AID-SMR200>3.0.CO;2-W
33. 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)

34. Larman, C., Basili, V.R.: Iterative and Incremental Development: A Brief History. IEEE Computer 36(6), 47–56 (2003), `http://doi.ieeecomputersociety.org/10.1109/MC.2003.1204375`
35. Lehman, M.M.: Software's Future: Managing Evolution. IEEE Software 15(01), 40–44 (1998)
36. Lehman, M.M., Belady, L.A. (eds.): Program evolution: processes of software change. Academic Press Professional, Inc., San Diego (1985)
37. Leymann, F.: Combining Web Services and the Grid: Towards Adaptive Enterprise Applications. In: Castro, J., Teniente, E. (eds.) First International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA 2005) - CAiSE Workshop, pp. 9–21. FEUP Edi cões (June 2005), `http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2005-123&engl=1`
38. Malek, S., Edwards, G., Brun, Y., Tajalli, H., Garcia, J., Krka, I., Medvidovic, N., Mikic-Rakic, M., Sukhatme, G.: An architecture-driven software mobility framework. Journal of Systems and Software 83(6), 972–989 (2010)
39. Malek, S., Mikic-Rakic, M., Medvidovíc, N.: A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems. In: Dearle, A., Savani, R. (eds.) CD 2005. LNCS, vol. 3798, pp. 99–114. Springer, Heidelberg (2005)
40. Mens, T.: Introduction and Roadmap: History and Challenges of Software Evolution. In: Software Evolution, ch.1. Springer (2008), `http://www.springerlink.com/content/978-3-540-76439-7`
41. Müller, H.A., Pezzè, M., Shaw, M.: Visibility of control in adaptive systems. In: Proceedings of Second International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008), pp. 23–27. ACM/IEEE (2008)
42. Murray, R.M., Åström, K.J., Boyd, S.P., Brockett, R.W., Stein, G.: Future Directions in Control in an Information Rich World. IEEE Control Systems 23, 20–33 (2003)
43. Object Management Group (OMG): Software & Systems Process Engineering Meta-Model Specification (SPEM), Version 2.0 (2008)
44. 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 Intelligent Systems 14, 54–62 (1999), `http://dx.doi.org/10.1109/5254.769885`
45. Osterweil, L.J.: Software processes are software too. In: Proceedings of the 9th International Conference on Software Engineering (ICSE 1987), pp. 2–13. IEEE Computer Society Press, Los Alamitos (1987), `http://portal.acm.org/citation.cfm?id=41765.41766`
46. Sawyer, P., Bencomo, N., Whittle, J., Letier, E., Finkelstein, A.: Requirements-Aware Systems. A Research Agenda for RE For Self-Adaptive Systems. In: 18th International Requirements Engineering Conference (RE 2010), pp. 95–103. IEEE (2010)
47. Seborg, D.E., Edgar, T.F., Mellichamp, D.A., Doyle III, F.J.: Process Dynamics and Control, 3rd edn. John Wiley & Sons (1989)
48. Shaw, M.: Beyond objects. ACM SIGSOFT Software Engineering Notes (SEN) 20(1), 27–38 (1995)
49. Shaw, M.: The role of design spaces in software design (2011) (submitted for publication)

50. Swanson, E.B.: The dimensions of maintenance. In: Proceedings of the 2nd International Conference on Software Engineering (ICSE 1976), pp. 492–497. IEEE Computer Society Press (1976), http://portal.acm.org/citation.cfm?id=800253.807723
51. Tanenbaum, A.S., van Steen, M.: Distributed Systems: Principles and Paradigms, 2nd edn. Prentice-Hall, Inc., Upper Saddle River (2006)
52. Villegas, N.M., Müller, H.A.: Context-driven Adaptive Monitoring for Supporting SOA Governance. In: 4th International Workshop on a Research Agenda for Maintenance and Evolution of Service-Oriented Systems (MESOA 2010). CMU/SEI-2011-SR-008, Pittsburgh: Carnegie Mellon University (2011), http://www.sei.cmu.edu/library/abstracts/reports/11sr008.cfm
53. Villegas, N.M., Müller, H.A., Tamura, G., Duchien, L., Casallas, R.: A Framework for Evaluating Quality-driven Self-Adaptive Software Systems. In: Proceedings 6th International Symposium on Software Engineering for Adaptive and Self-managing Systems (SEAMS 2011), pp. 80–89. ACM, New York (2011), http://doi.acm.org/10.1145/1988008.1988020
54. Villegas, N.M., Müller, H.A., Tamura, G.: Optimizing Run-Time SOA Governance through Context-Driven SLAs and Dynamic Monitoring. In: 2011 IEEE International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA 2011), pp. 1–10. IEEE (2011)
55. Villegas, N.M., Müller, H.A., Muñoz, J.C., Lau, A., Ng, J., Brealey, C.: A Dynamic Context Management Infrastructure for Supporting User-driven Web Integration in the Personal Web. In: 2011 Conference of the Center for Advanced Studies on Collaborative Research (CASCON 2011), pp. 200–214. IBM Corp., Markham (2011), http://dl.acm.org/citation.cfm?id=2093889.2093913
56. Vogel, T., Giese, H.: Adaptation and Abstract Runtime Models. In: Proceedings of the 5th ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010), pp. 39–48. ACM (2010), http://portal.acm.org/citation.cfm?id=1808984.1808989
57. Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Incremental Model Synchronization for Efficient Run-Time Monitoring. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 124–139. Springer, Heidelberg (2010), http://www.springerlink.com/content/1518022k168n5055/
58. Vromant, P., Weyns, D., Malek, S., Andersson, J.: On interacting control loops in self-adaptive systems. In: Proceedings of Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011), Honolulu, Hawaii (2011)
59. Weyns, D., Malek, S., Andersson, J.: On decentralized self-adaptation: lessons from the trenches and challenges for the future. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2010, pp. 84–93. ACM, New York (2010), http://doi.acm.org/10.1145/1808984.1808994
60. Dahm, W.J.A.: Technology Horizons a Vision for Air Force Science & Technology During 2010-2030. Tech. rep., U.S. Air Force (2010)

A Design Space for Self-Adaptive Systems

Yuriy Brun[1], Ron Desmarais[2], Kurt Geihs[3], Marin Litoiu[4],
Antonia Lopes[5], Mary Shaw[6], and Michael Smit[4]

[1] Computer Science & Engineering, University of Washington, Seattle WA, USA
brun@cs.washington.edu
[2] University of Victoria, Vancouver, Canada
rd@uvic.ca
[3] EECS Department, University of Kassel, Kassel, Germany
geihs@uni-kassel.de
[4] York University, Toronto ON, Canada
{mlitoiu,msmit}@yorku.ca
[5] Department of Informatics, University of Lisbon, Lisboa, Portugal
mal@di.fc.ul.pt
[6] Institute for Software Research, Carnegie Mellon University, Pittsburgh PA, USA
mary.shaw@cs.cmu.edu

Abstract. Self-adaptive systems research is expanding as systems professionals recognize the importance of automation for managing the growing complexity, scale, and scope of software systems. The current approach to designing such systems is ad hoc, varied, and fractured, often resulting in systems with parts of multiple, sometimes poorly compatible designs. In addition to the challenges inherent to all software, this makes evaluating, understanding, comparing, maintaining, and even using such systems more difficult. This paper discusses the importance of systematic design and identifies the dimensions of the self-adaptive system design space. It identifies key design decisions, questions, and possible answers relevant to the design space, and organizes these into five clusters: observation, representation, control, identification, and enacting adaptation. This characterization can serve as a standard lexicon, that, in turn, can aid in describing and evaluating the behavior of existing and new self-adaptive systems. The paper also outlines the future challenges for improving the design of self-adaptive systems.

Keywords: adaptive, self-adaptive, design, architecture.

1 Introduction

Designing a self-adaptive software system involves making design decisions about how the system will observe its environment and choose and enact adaptation mechanisms. While most research on self-adaptive systems deals with some subset of these decisions, to our knowledge, there has been neither a systematic study of the design space nor an enumeration of the decisions the developer should address. Developers draw from their own backgrounds to make decisions about self-adaptive system design. At worst, the design is predicated on modifying the existing system until it appears to work.

A *design space* is a set of decisions about an artifact, together with the choices for these decisions. A design space serves as a general guide for a class of applications,

R. de Lemos et al. (Eds.): Self-Adaptive Systems, LNCS 7475, pp. 33–50, 2013.

enumerating decisions and alternatives to provide a common vocabulary for describing, understanding, and comparing systems. A designer seeking to solve a problem may be guided by the design space, using it to systematically identify required decisions, their alternatives, and their interactions.

Intuitively, a design space is a k-dimensional Cartesian space in which design decisions are the k dimensions, possible alternatives are values on those dimensions, and complete designs are points in the space. In practice, most interesting design spaces are too rich to represent in their entirety, so their representations capture only the principal decisions as dimensions. Also in practice, the design dimensions are neither independent nor orthogonal, so choosing an alternative for one dimension may affect other dimensions and preclude or make irrelevant other decisions [3, 16, 20]. Creating a design space leads to creating rules, guidelines, and best practices that identify good and bad combinations of choices.

In this paper, we outline a design space for self-adaptive systems with five principal clusters of dimensions (design decisions). The clusters are observation, representation, control, identification, and enacting adaptation. Each cluster provides additional structure in the form of questions a designer should consider. These questions are not necessarily novel themselves; indeed, some of them are fundamental to self-adaptive systems. Rather, we hope our enumeration and organization will help formalize and advance the understanding of self-adaptive system design as a whole. The formulation as a design space is intended to explicitly help design systems. We do not present this as a final and definitive design space; further work on expanding and refining this design space is necessary and appropriate.

The remainder of this paper is organized as follows. Section 2 identifies related work on characterizations and models for self-adaptive systems, and discusses how presenting a design space is a novel contribution. Section 3 presents (1) a basic architecture and terminology and (2) an example system. Section 4 describes the design space and Section 5 places the example system in that space. Section 6 outlines future challenges for defining the self-adaptive system design space. Finally, Section 7 summarizes our contributions.

2 Related Work

Examples of self-adaptive systems include autonomic computing systems and self-managing software systems. This broad term includes self-configuring, self-healing, self-adapting, and self-protecting. IBM has identified a four-stage cycle for autonomic computing, called MAPE-K: Monitor, Analyze, Plan, and Execute, with a Knowledge Base common to all components (see Figure 1) [12, 14]. Others have referred to this cycle as the collect / analyze / decide / act cycle [7]. The architectural model that employs such a cycle implements a feedback loop with the Controller comprising of the four stages of MAPE-K. This Controller uses measured output from sensors monitoring the managed system to choose control signals to send to the managed system. The Monitor stage uses sensors to measure key attributes, usually related to the current performance and load of the system. The Analyze stage identifies any metrics not within tolerances (or violating some type of defined rule) and attempts to identify the cause or source of

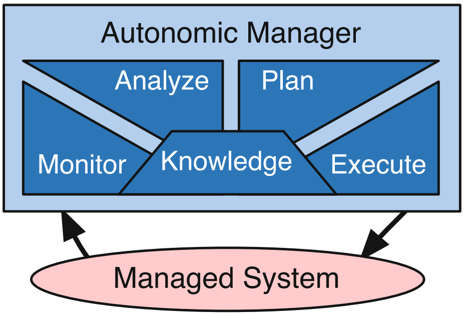


Fig. 1. The MAPE-K model (based on [14])

the problem. In the Plan stage, the system reacts to the fault (or generally, the results of the Analyze stage) by identifying a set of actions that remedy the situation. These actions are implemented in the Execute stage via actuators that act on the managed systems. All the stages make use of a Knowledge Base.

Several partial approaches to identifying and representing design spaces for self-adaptive software systems are available. These existing approaches have their roots in both software engineering (e.g., [7, 15]) and control theory (e.g., [4, 22]).

Andersson et al. [1] defined a set of modeling dimensions for self-adaptive software systems. The dimensions were organized into four groups, including the self-adaptive goals of the system, the causes or triggers of self-adaptation, the mechanisms used to adapt, and the effects of those mechanisms on the system. Their dimensions have some overlap with our dimensions; they and we both explicitly encourage the addition of more dimensions. Though parts of our work extend theirs, and we consider their work important and relevant, we offer a redevelopment with important differences that are in part related with the differences between modeling and design:

First, their dimensions are appropriate for classifying an existing adaptive system, while ours are more appropriate for designing a system based on a set of requirements. Their categories could be said to be driven by observation; that is, the inputs, the causes, and the effects. We make explicit the design decisions regarding how we observe and represent the inputs, how these inputs are translated into adaptation triggers, and what the capabilities and limits of our adaptation mechanisms are.

Second, our work aims at a higher level of abstraction. Some of their dimensions would be implementation decisions in our design space. For instance, whether one or several components are responsible for adaptation is a design dimension in their work.

Third, while there is some overlap among the categories of dimensions, each makes different exclusion and inclusion decisions. They view the adaptation goal as one of the major categories (having 5 dimensions). We assume the goals (called adaptation targets) are dynamic, multiple, and specified at run-time not design time (and therefore not explicitly included in the design space). However, our design space is still applicable to static, singular adaptation targets. We identify several key dimensions not included in their work, in particular those related to modeling, representation, and control loops. For example, though perhaps not essential to a working implementation, an explicit conceptual understanding of how the control loops are orchestrated improves the developers' understanding. We include assessing what is possible: what can be observed, what can be adapted, whether the managed system provides appropriate mechanisms to achieve the desired control, etc.

Kramer and Magee [15] outline three tiers of decisions the developer must make — ones that pertain to goal management, change management, and component control. Finally, Brun et al. [4] discuss the importance of making the self-adaptation control loops explicit during the development process and outline several types of control loops that can lead to self-adaptation.

Villegas et al. [22] describe a characterization model for approaches to self-adaptive systems that include a survey of existing self-adaptive systems. They systematically identify what these systems use for each element of self-adaptation (these elements are described here in Section 3). Their results support our claim that current systems are often designed using an ad-hoc approach. This post-hoc analysis of past implementations is interesting and useful; their focus is on evaluating and characterizing self-adaptive systems post-implementation, rather than guiding design decisions at design time.

Taxonomies for self-adaptive systems have also been proposed. For example, Brake et al. [2] and Ghanbari et al. [10] introduce and discuss a taxonomy for performance monitoring of self-adaptive systems together with a method for discovering parameters in source code. Checiu et al. [6] formally defined controllability and observability for web services and showed that controllability can be preserved in composition.

3 Self-Adaptive Systems

This section describes (1) a basic architecture and terminology for self-adaptive systems used as a working definition throughout the paper, and (2) a web application that illustrates this terminology and will serve as a running example throughout the remainder of this paper.

We separate self-adaptive systems into two elements: the Managed System, which is responsible for the system's main functionality, and the Adaptation System, which is responsible for altering the Managed System as appropriate. Figure 2 shows these two elements linked by the flow of system indicators and adaptations. The elements inherent to the managed system (that is, the things that would exist even if it were not adaptively managed) such as the inputs and the environment are captured and used by the Adaptation System. The Adaptation System produces adaptations that impact the Managed System.

Example. To illustrate these concepts, consider the following example which will be used throughout the paper to illustrate the design space. This generic example is representative of a large class of self-adaptive systems.

Consider a web application running on one or more web servers. This application must support an arbitrary number of users with a response time below a predefined threshold. The application accesses a database, but this is only a potential bottleneck when certain intensive features are used, like the recommendation engine. The application must also be available for a predefined minimum percentage of time, except for defined maintenance windows. The operational and environmental conditions (e.g., the number of users) change during execution. Elements of the application or the hardware

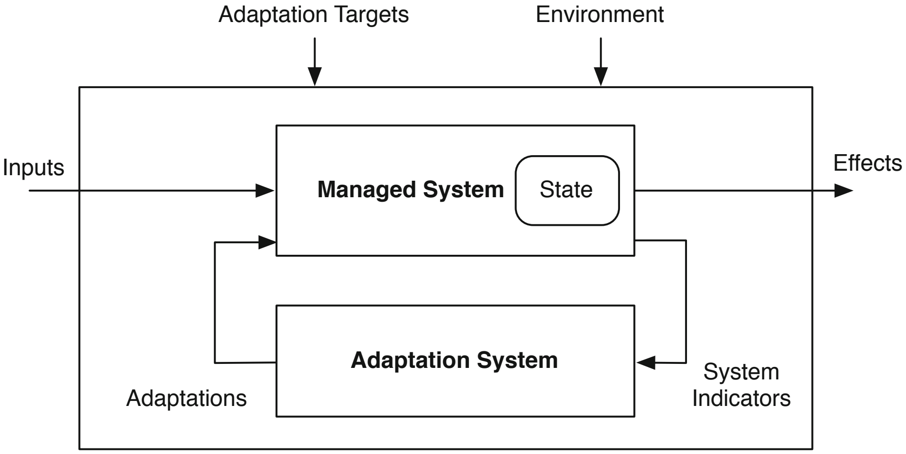


Fig. 2. Elements of a self-adaptive system. The arrows indicate information flow

server may fail at any time. The web application proper (Managed System) is augmented with an adaptation driver (Adaptation System) that will monitor the application and select and apply the necessary changes at runtime.

Definitions. The information used by the Managed and Adaptation System is described in more detail below.

- *Adaptation Targets* define what the self-adaptation should achieve. Adaptation targets are not necessarily end user requirements; they could be design goals derived by determining what is required to meet end user requirements in an optimal manner. For instance, in our example, the end user requirement *response time below a threshold* could result in an adaptation target outlining how to achieve this requirement with minimum resources. These targets can be expressed in different ways depending on the domain and the design decisions.
- *Effects* are what Managed Systems produce per their functional requirements; however, included are *system indicators* (of functional or non-functional properties) which can be evaluated to determine how compliant the managed software is with the adaptation targets, and are part of the input to the Adaptation System. In our example, the *response time* of the web application would constitute an effect parameter. The evaluation process may be more complex than comparing a momentary value to an established threshold. Again in the example, the *availability over time* effect requires capturing availability information over time and calculating an average over a sliding window of time.
- *Adaptations* are the actions taken by the Adaptation System. One type of adaptation is *parametric adaptation*, the adjusting of tuneable parameters to attain the desired behavior from the Managed System. The correct tuning for these parameters

is computed by the Adaptation System by taking into account the System Indicators, Environment and State. In the web application, a tuneable parameter might be the *number of threads* used to service user requests. By adjusting the number of threads, the State is changed (specifically the thread queue length state) and the Effect and System Indicators are affected (response time). Another type is *architectural adaptation* (or composition adaptation), for instance exchanging one component for another. In our example system, depending on how the change is effected, changing the type of recommendation engine would be an example of exchanging one component for another.

- *State* is the representation of the internal state of the Managed System, and is comprised of a collection of state parameters that characterize or model the Managed System. The state of the Managed System is affected by the Inputs and Environment streams, and plays a role in determining the Effects and System Indicators. In our web application example, the *length of the thread queue* serving user requests could be a State parameter. Other parameters could be identified based on their ability to accurately characterize or model what is happening inside the Managed Application. The State used in an adaptive system is highly dependent on which System Indicators can be measured and with what accuracy, a challenge discussed further in the Observation cluster (Section 4.1).
- *Environment (or perturbation)* indicators are external to the self-adaptive software and the Managed System, but have a direct or indirect influence on the State, Effects, and System Indicators. For our example, the *workload* (e.g., the number of users accessing the application running on the web server or the frequency of their requests) is an environment indicator. These indicators typically cannot be controlled (directly or indirectly).

4 Dimensions of the Design Space

We describe the design space for self-adaptive systems using various *dimensions*, each defined by a design question that admits more than one possible answer (the design decision). The dimensions are from our own experience designing self-adaptive systems, and can and should be extended with the experience of others. To manage the complexity, we group dimensions into *dimension clusters*, such that each cluster represents a particular category of concern. The clusters, shown in large font in Figure 3, represent the design space. There is natural overlap between the clusters (indicated by dimensions affecting multiple clusters) and dependence between some dimensions (indicated by similarity of color). The clustering helps manage the complexity of the design space, but it is also possible to consider the dimensions independently of their clusters.

The *Observation* cluster includes questions related to what is monitored by the Adaptation System, when and how often monitoring occurs, and how states are determined based on the monitored data. The *Representation* cluster is concerned with the runtime representation of adaptation targets, inputs, effects, system indicators, and states. The *Control* cluster is concerned with the mechanisms through which a solution is enacted.

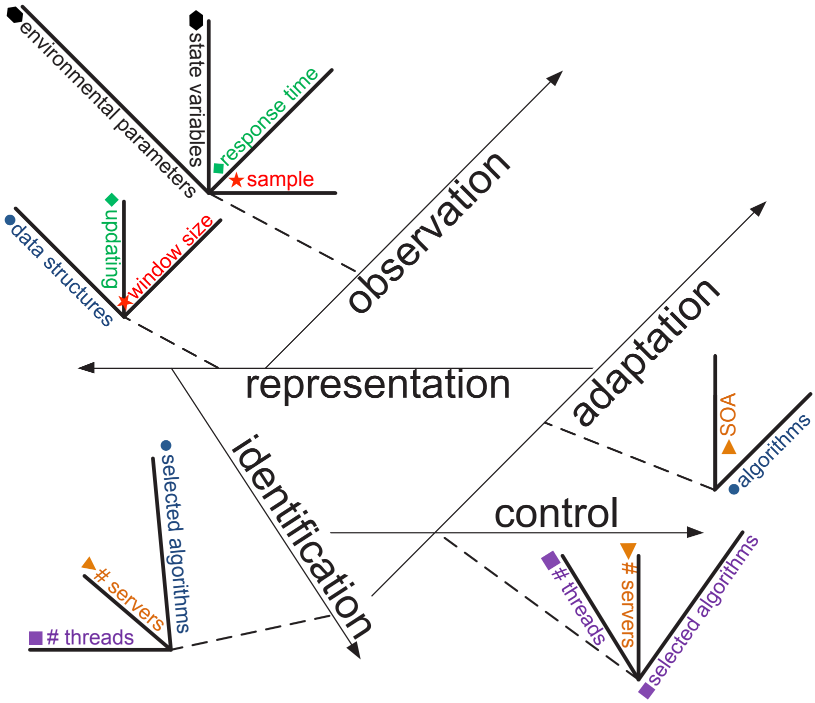


Fig. 3. The dimensions of the self-adaptive system design space can be grouped into five clusters, shown here with a sampling of properties for our example system. Intersections among the dimension lines indicate potential overlap. Sample properties with bulleted text in the same colour represent dependencies. For example, the *window size* chosen in representation depends on the *sample* size available in observation.

While these three dimensions are somewhat independent, they have more significant overlap with the final two. The *Identification* cluster is concerned with the possible solution instances that can achieve the adaptation targets. Finally, the cluster of *Enacting Adaptation* concerns how adaptation can be achieved.

Table 4 details the clusters and each cluster's dimensions. For each cluster, we name the key dimensions (in the form of questions) and by way of example suggest answers to these questions. When using the design space as a guide for software design, it is important that the answers be more specific and implementation-focused. For instance, for the question "What triggers observation of metric m?" a general answer is "periodically, on a timer," whereas a more-specific answer is "every k seconds, for $k \in [1,2,\ldots,1000]$ sec." The level of appropriate abstraction depends on the design space's use. Section 5 steps through the dimension clusters answering design questions for the sample application described in Section 3.

While we hope our enumeration will help formalize and advance the understanding for self-adaptive system design, it is not intended to be complete and further work on expanding and refining this design space is necessary and appropriate.

Observation
– what can be observed?
– what information will the system observe?
– what information about the external environment and about the system itself will need to be measured and represented internally?
– how will the system determine that information?
– what triggers observation?
– what triggers adaptation?
– how is uncertainty in the observations handled?

Representation
– what information is made available to the components of the self-adaptive system?
– how is this information represented?
– when and how is the information updated?

Control
– does the system provide enough operations/parameters to achieve the desired control?
– how does the system decide what and how much to change to modify its behavior?
– how will the control loops be orchestrated?
– what to adapt?
– how to adapt?
– when to adapt?

Identification
– what are possible solutions for a given set of adaptation targets?
– which solutions are or will be supported?
– what are relevant domain assumptions and contexts for each solution?
– what are the required observed and control parameters for each solution?

Enacting Adaptation
– is the adaptation mechanism represented explicitly or implicitly in the architecture?
– how is adaptability supported?
– how will failures of the adaptation mechanism be handled?
– what is the cause of adaptation?

Fig. 4. Dimensions of the design space and their primary questions, organized into clusters

4.1 Observation

The observation cluster includes dimensions covering design decisions regarding what information is observed by the Adaptation System. Generally, the dimensions in this cluster are related to the question "How do we gather information about the Managed System (System Indicators and States) and its Environment?" This cluster is related to the representation cluster: observations gathered with no mechanism for internal representation are superfluous, and internal representations not updated by observations are potentially serious problems. (Although we acknowledge that some reactive systems may act on observations directly, without internalizing them in a representation.)

At a conceptual level, the information gathered in a self-adaptive system is intended to be a proxy measure for one or more higher-level constructs, such as workload, performance, or safety. The designer chooses system and environmental properties that they expect to be good predictors of these constructs. However, the ability to measure these properties is often limited. For instance, to decrease the overhead of monitoring, periodic samples are taken, and intermediate values are inferred. If physical elements are involved (e.g., CPU temperature, server room temperature, voltage, etc.), there may be limits on where a sensor can be placed, and on the accuracy of that sensor. The designer must be aware of what value the self-adaptive system will be using. If not precisely the CPU temperature, is it the temperature at some point near the CPU where a sensor can be installed? If so, can the value returned by the sensor be trusted? Or should it instead be a value derived from the sensor but corrected by calibration and averaging? If so, what inaccuracies are introduced by the time lag waiting for these measurements?

We wrap up these questions into a single question: "what *can* be observed?" For any system property of interest (perhaps because that's a target, perhaps because it's used in a model), the designer must understand if it is possible to observe directly, if it must be indirectly estimated by inference or aggregation, etc. In short, the designer must distinguish between "what the system wants to measure" and "what the system can measure and infer from measurements." This makes it explicit that the self-adaptive system relies on the ability of the observations to actually estimate the desired system property, and may even impose proof obligations on the designer to show that what the system is capable of measuring is an acceptable proxy for the higher-level construct that is conceptually important.

A key design decision about self-adaptive systems is "what information *will* the system observe?" In particular, "what information about the external environment and about the system itself will need to be measured and represented internally?" The self-adaptive system may require awareness of the context within which the managed system is operating (e.g., [9]). Answering these questions requires an understanding of how the information will be used, which may require other design decisions to be made first.

Given the set of information the system observes, another important design decision is "how will the system determine that information?" The system could make direct measurements with sensors, use measurements provided by an already instrumented system, infer information from a proxy, extrapolate based on earlier measurements, aggregate knowledge, etc. Some of the measurements can be made implicitly, e.g., by inferring them from the state of the system or success or failure of an action. Others will require aggregation or reasoning over time or events, or be in some way combined with previous measurements. The answer may be different for each metric being observed.

Knowing what to observe and how it is observed, the next dimension is when to observe. There are actually two timing questions: "what triggers observation?" and "what triggers adaptation?" The system could be continuously observing or observation could be triggered by an external event, a timer, an inference from a previous observation, deviation or error from expected behavior, etc. Thus, the observation can happen at a fixed delay, on-demand, or employ a best-effort strategy. The same decisions relate to when the adaptation triggers. Again, this question may need to be answered differently for each item in the set of information being observed.

A final set of decisions dealing with observation is "how is uncertainty in the observations handled?" Answers could include filtering, smoothing, and redundancy, or perhaps the system might not have a specific strategy to deal with noise and uncertainty.

4.2 Representation

The representation cluster is concerned with design decisions regarding how the problem and system are represented at runtime. To enable mechanisms of adaptation, key information about the problem and system may have to be made available at runtime. As mentioned, this cluster is closely related to the observation cluster, and has ties to the remaining clusters that make use of this representation.

A key decision in this cluster is "what information is made available to the components of the self-adaptive system?" Answers include different aspects of the adaptation targets, existing domain knowledge, and observations of the environment and Managed System that are found necessary and sufficient for self-adaptive system to operate.

Another design decision in this cluster relates to choices of internal representation. "How is this information represented?" is meant to guide the designer to the representation that best matches the adaptation targets and the nature of the problem. Choices include explicit representations such as graph models, formulae, bounds, objective functions, etc., or implicit representations in the code.

Since some of the information in the representation is dynamic — primarily observations of the environment and the Managed System, but potentially also the adaptation targets or the knowledge model — it is necessary to have an approach to update the representation: "when and how is the information updated?" This dimension prompts a decision regarding when updates will occur (a fixed delay, on-demand, best effort, etc.), and how these updates occur (push? pull? with notification to concerned components?). Of course, static information will not be updated; another important design decision is what information is static.

4.3 Control

This cluster is concerned with the mechanisms whereby system execution changes the Managed System in some way to bring it in line with the adaptation targets.

Perhaps the most important question regarding this cluster is "does the system provide enough operations or parameters to achieve the desired control?" The designer has the obligation to show that the choices made in the Identification cluster are sufficient to achieve the desired adaptation targets.

One dimension in this cluster is "how does the system decide what behavior to modify and by how much?" Deciding "what" will change depends on knowing what adaptations (control inputs, architectural changes, deployment changes, etc.) are available, and then identifying those that will modify the system appropriately. The amount of the change can be a predefined constant value, or can be proportional to, or a function of the deviation from the desired behavior, or of another factor. In some cases, the change is a sum of three terms, a control technique known as *PID*: a component proportional with the control error (*P*, roughly the error at the time), a component proportional with the derivative of the error (*D*, roughly the rate of change of the error) and another proportional with the integral of the error (*I*, roughly the accumulated error over time). In addition to the PID controller, other control-systems analogies can also be used. In situations with the target of modifying along several requirements, optimizing a utility function may be appropriate. Complex modifications may require planning and re-planning.

Adaptation is realized through feedback loops and complex software systems require multiple loops to adapt [4]. A key decision is "how will the control loops be orchestrated?" Possible answers depend on the structure of the system and the complexity of the adaptation targets. Composition patterns include series, parallel, multi-level (hierarchical), nested, and independent loops. The control loops may not be explicit in all self-adaptive systems, but some understanding of the nature of the loops (feedback, feedforward, etc.) is important to guide the design.

Control loops need to be further augmented with actuation decisions. The design questions "what to adapt?" can have several answers. We can adapt parameters, representation, resource allocation, component and connector selection, component and connector synthesis, sensor and actuator augmentation, component deployment, etc. Once the actuators are selected, the next question becomes "how to adapt?" This decision provides concrete actuation methods, including abort, modify data, call procedure, start new process, etc. Like observation and representation, there is a time dimension in control. "When to adapt?" can lead to choices such as: immediately, on a time scale, with fixed delay, continuous change, lazy, on-demand, best effort, etc.

4.4 Identification

The identification cluster defines the possible solutions a self-adaptive system instance can assume when it adapts. The solutions can cover changes in system structure, in its behavior, or in a combination of both.

The first dimension in this cluster identifies adaptation solutions: "what are possible solutions for a given set of adaptation targets?" Finding a discrete set of possible structural changes, states, parameter values, etc. is the main concern. Changes can include parametric adaptations where some tuneable parameter is adjusted, architectural (compositional) adaptations where software components or their connections are replaced or updated, and deployment adaptations where the deployment of the adaptation system is changed (e.g., moving it to the public cloud). Not all solutions will be supported at runtime. Answering the question "which solutions are or will be supported?" narrows the list to a subset that will be available at runtime.

Another set of dimensions identifies runtime contexts and feedback loop parameters for a particular set of solutions. The "what are relevant domain assumptions and contexts for each solution?" design question provides the runtime context for the available self-adaptive solutions. "What are the required observed and control parameters for each solution?" identifies prerequisites; this information should be taken into consideration in the observation and representation clusters. It also provides the necessary information to enable the changes in feedback loop components when switching from one solution to another.

4.5 Enacting Adaptation

The Enacting Adaption cluster includes the adaptation mechanisms, how they are triggered, how they are supported, and how failure is handled; it is the closest of the clusters to the implementation solution.

A first question to ask is "is the adaptation mechanism represented explicitly or implicitly in the architecture?" Some feedback loops can be represented explicitly in the architecture whereas others are intrinsic to the system functionality and therefore represented implicitly. A self-adaptive system can also be a hybrid of explicit and implicit loops.

The answers to the question "how is adaptability supported?" might depend on issues not related to adaptivity itself but on other system goals, such as maintainability and reliability. This could be answered in terms of the architecture of the Managed System (a plugin architecture, web services style, etc.) or in terms of the adaptations available on the Managed System. Closely related to the previous design question is "how will failures of the adaptation mechanism be handled?" e.g., try again, adopt another mechanism, etc.

Design decisions in all clusters will be affected by what causes adaptation, so it is important to answer the question of "what is the cause of adaptation?" Answers include: non-functional requirements (these often use control-theoretic concepts and relate to response time, throughput, energy, consumption, reliability, fault tolerance), behavior, prevention of undesirable events, maintaining state, dealing with anticipated change, dealing with unanticipated change, etc.

5 Using the Design Space

The design space is intended to be useful for tasks such as describing the behavior of existing self-adaptive systems using a shared standard lexicon, characterizing or creating a taxonomy for self-adaptive systems, evaluating a self-adaptive system using the design space as a checklist, comparing several systems or designs, and designing a new self-adaptive system. The primary intended use for the design space is to serve as a guide when designing a self-adaptive system. The questions of each dimension either can be answered throughout the design document or can be explicitly separated into their own chapter.

To illustrate this use of the design space, we show here the design decisions for our sample managed web-server system (Section 3), with the adaptation target of achieving response time within a given limit for 95% of requests, while controlling costs. Note we have not explicitly shown these design decisions are the right ones; that is, we show what a point in the design space looks like, but do not make a claim that it is the correct point (or one of the several possible correct points).

5.1 Observation

The Adaptation System will observe the effect Response Time; the internal state parameters Actual Number of Threads, Thread Queue Length, CPU Utilization; and the environmental indicators of workload Request Arrival Rate and Number of Users.

The Adaptation System also observes the state of the available control inputs — number of threads, number of servers, and the algorithm used by the Recommendation Engine.

The effect and environmental indicators are captured at the application boundary (measuring incoming requests and outgoing responses); the state parameters and inputs are captured using sensors built into the application itself. Observations at the application boundary are made after every k requests and reported in the aggregate (median, mean, standard deviation) over those k requests, with a configurable k (default is $k = 1000$). The state parameters are observed every j seconds, with a configurable j (default $j = 180$). Control inputs are observed, and can be changed every i minutes, with i configurable (default $i = 10$). Uncertainty in the observations is not handled by the observation and monitoring components, though we note that high variance in the data can be measured by observing its standard deviation.

5.2 Representation

The information observed is available, both current and historic for a configurable length of time. Other than the information discussed in the Observation cluster above, the system also keeps track of

- 95th percentile response time over various sliding windows of time (contract period, last day, last hour) to measure compliance with the adaptation target,
- the current cost of the configuration (computed based on control inputs and current architecture),
- past configuration and workload pairs that have met the adaptation target, and
- the current target required by the adaptation target.

The software components can access the observed information though internal data structures. Known-good configuration and workload pairs are represented in a machine-readable knowledge base, translated into a set of logical rules that can be used to produce recommendations or to power an expert system. (For the purpose of this example, we leave aside the details of the expert system design and treat it as an oracle loosely coupled to the Adaptation System.) The adaptation targets are expressed as objective functions.

The adaptation targets must be updated manually (not automatically). The rest of the represented information is dynamic and will be updated as follows:

- Rules are added to the knowledge base without notification to any components. The intent for is for the rules to considered in the next recommendation produced by the knowledge base.
- The software data structures used to store and access observed information ensure that any request for information will always return the most up to date information available. Such requests will block if there is a pending or overdue attempt to update. The data structure is updated as the information is observed (see the Observation cluster above).

5.3 Control

The controls available to our system are: adding and removing a thread, and adding and removing a server. The available architectural change is swapping out the recommendation engine component. The number of threads that can be added is limited by the resources available on the currently deployed servers. The number of servers is theoretically not bounded. Resources scale non-linearly: adding a thread for which there is enough capacity is quite inexpensive, but adding a whole server is more expensive. (We assume a utility computing model, such as the cloud, where additional scaling — adding a rack or adding data center capacity — is not required.) The Recommendation algorithm can be either highly personalized (database intensive) or generic (with low resource usage). Our example does not allow for database replication, so we do not consider the complexity of added revenue from a personalized recommendation engine. Threads can be removed one at a time and are added proportionally to how overloaded existing threads are. Servers are added and removed one at a time. The decision to change is based on the optimization of the target adaptation objective function and the recommendation of the expert system and knowledge base.

For this simple example, there is one primary feedback loop that manages response time. There are simple, nested feedback loops that identify corrective actions based on the current state of the control inputs. If we included other adaptation targets (e.g., managing uptime), there would be parallel feedback loops.

The actuator for adding a thread launches a lightweight process and adds an entry to the load balancer. Removing a thread involves identifying the oldest lightweight process, removing its entry from the load balancer, stopping it safely, and monitoring to ensure the stop command succeeds (in necessary, the process can be killed). The actuator for adding a server involves a utility-computing command to deploy a new server from a given machine image and to trigger the thread actuator to add new threads. Server removal requires safe termination of the threads and utility-computing commands. Changing the recommendation engine algorithm involves making a call to the web application API, then updating the internal representation of the Managed System to note the reduced resource usage that will enable more threads per server. All actuations of adaptations occur immediately.

5.4 Identification

Adaptation solutions define architectural changes and ways of influencing the state and effect variables through changes in control inputs. Alternative solutions can be explicitly represented in the design or implicitly represented in the algorithms and functions.

In the web-server example, changes will take place when:

- the expert system makes a recommendation, and
- the feedback loop identifies that additional resources are required, based on observations of the state of the application and the environment.

The self-adaptive solutions supported at runtime are combinations of:

A) change to the low-cost Recommendation Engine,
B) change to high-cost Recommendation Engine,

C) add threads to a server,
D) remove threads from existing servers, and
E) add a server.

The specific parameters and contexts for effecting each solution above, respectively:

A) When the database layer is the identified bottleneck (CPU utilization on the application server is low; thread queue is high).
B) When resource utilization is low and less than four application servers are in use.
C) When additional capacity is needed to ensure the adaptation target is met and when sufficient memory and CPU resources are available on a server but requests need to be processed faster.
D) When resource utilization is low and less than four application servers are in use.
E) When sufficient memory and CPU resources are not available on a server, requests need to be processed faster, and the utility cost of deploying a new server is less than the utility cost of violating the adaptation targets for the projected length of time the workload will remain high.

An important mechanism of evaluating adaptation solutions at runtime is the representation of the solution space through a quantitative model. Quantitative models are quality attribute-specific, and as such performance, reliability, security and availability have different models. For our example, we can choose from four major classes of performance models [18]. Queuing models can be implemented through simulation [21] or analytical models [17] and are easy to reconfigure at runtime for vertical and horizontal scaling of applications. Dynamic models such as regression models allow for a synthesis of the control using classic system control techniques. Policy models are a set of rules that capture the designer experience or can be discrete representations of more complex models. A Markov Decision Process model can be used to compute optimal reactions to state changes, combining observations on the system (for example, an observation of the system state) with assumptions about the rate of changes of state that are expected in the future.

5.5 Enacting Adaptation

The adaption mechanisms are built into the Managed System, and the observation and other elements of the feedback loop are explicitly designed and implemented directly in the code. The feedback loop is implicit. The Managed System is built with a service-oriented API and instrumentation to support being managed by the Adaptation System. When the provided adaptation mechanisms fail, the system will retry the adaptation at the next control interval. If the failure repeats, the system will attempt designated fallbacks (e.g., if adding a thread fails, the system will add a server). When the fallbacks are exhausted, the system will generate a notification that intervention is required and will retry the adaptations until it either unnecessary, works, or halted by the user.

The adaptation is caused by the expert system and through measurements on the Managed System.

6 Self-Adaptive System Design Space Challenges

While the design space described above helps formalize and advance the understanding of self-adaptive system design, it is not complete.

The main remaining challenge is to infuse a systematic understanding of the alternatives for adaptive control into the design process. Further, the trade-offs among the choices and quantitative and qualitative implications of the design decisions on the overall adaptation quality need to be investigated. The clusters of dimensions described here pose the questions that need to be answered by designers of self-adaptive systems. A complete solution, however, will also include the possible answers and a systematic way of evaluating the trade-offs of each choice (such as the SEI approach to architectural design [13]).

Another remaining challenge is to expand and refine the dimensions in each cluster. At the same time, since the dimensions are not independent, the dependencies among the dimensions and choices need to be understood. Such proper understanding can narrow the search through the design space, improve the efficiency of the design process, and reduce the design complexity. Validating the dimensions and their constraints against real-world examples can serve as the framework for describing alternatives and as a checklist to help designers avoid leaving out critical decisions.

This initial description has largely set aside the question of evaluating proposed designs. While the design space can be used as-is to ensure a design has considered the various dimensions, further work is required to help designers evaluate if their choices will lead to a system that achieves the desired control. Not all the points in the design space correspond to systems that will solve a given problem. How does the designer decide whether a given set of decisions (i.e., a point in the design space) suffices?

Bridging the gap from a generalized design space to an implementation is a complex challenge. The design space helps guide decisions but offers little guidance on the implementation of those decisions. Design patterns, architectural patterns, middleware, and frameworks can help bridge this gap. In particular, design patterns are solution templates designed to be reused [8]. The choices for each question in the design space could map to specific design patterns. Ramirez and Cheng [19] identify several design patterns specific to self-adaptive systems. Architectural patterns are to design patterns but target a higher level of abstraction, serving as blueprints that can be used and modified to suggest typical architectural solutions to common problems [5]. Middleware and frameworks and are more concrete and provide partial implementation, such as implementations of common tasks, common solutions, and common architectures for self-adaptive systems. As there are many proposed frameworks, architectures, and middleware for self-adaptive systems, selecting an appropriate approach based on choices made in the design space is an open, and non-trivial research problem.

For existing systems that need to be re-engineered to be self-adaptive, the design space may be even more constrained and complex than the space we have explored here. Extra dependencies and constraints may eliminate potential solutions in the design space. Tools can likely help understand this space, and the constraints.

Since complex self-adaptive systems will have more than one control loop, it is imperative that the common ways in which control loops interact be defined, along with patterns or common templates for handling these cases. For example, Hellerstein et al. [11] describes two systems with multiple feedback loops. The first system is a self-tuning regulator, which adapts the controller's parameters, which, in turn, manage a plant. In this case, the plant (Managed System), does not need to be modified because the self-tuning regulator can determine the Managed System's internal state from its effect. The second system is a gain scheduler. For this system, the output of the Managed System is not sufficient to determine its state and the Managed System must allow itself to modified to provide the necessary scheduling variables. Thus, a remaining challenge is to better understand the control-loop-related questions that need to be answered in designing systems with multiple forms of feedback.

7 Conclusion

Before designing a self-adaptive software system, it is valuable to explore the design space of such systems and to understand the dimensions of the space. Further, the understanding of the design space can increase maintainability and interoperability of deployed systems and facilitate evaluation and enhancement of existing and proposed systems. In this paper, we have described the design space of self-adaptive systems in terms of five clusters of dimensions: observation, representation, control, identification, and enacting adaptation. Each dimension is a question a designer must answer. We have also proposed possible answers for each dimension question and discusses dimension dependencies. Finally, we have illustrated the design space with a self-adaptive web-server system.

This paper represents the next step in what we hope will be a confluence of several lines of research on the design of self-adaptive systems. The design space we have outlined here is not complete and not exhaustive. Further work defining and understanding the design space — primarily by adding additional key dimensions and possible answers to existing questions — will increase its utility. Understanding the trade-offs, dependencies, and best practices for the design space will further enrich the experience of self-adaptive system developers. Finally, establishing paths from the design space to the system implementation, via design and architectural patterns, frameworks, and middleware, will assist designers and developers.

We believe that understanding the design space of self-adaptive systems and following a more-systematic approach to such system design will ultimately improve the outcomes of self-adaptive systems research.

Acknowledgments. We thank Gregor Engels, Jeff Magee, and John Mylopoulos for contributing ideas to the discussions that led to this paper.

References

1. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Modeling Dimensions of Self-Adaptive Software Systems. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 27–47. Springer, Heidelberg (2009)
2. Brake, N., Cordy, J.R., Dancy, E., Litoiu, M., Popescu, V.: Automating discovery of software tuning parameters. In: Proceedings of the 3rd International Workshop on Software Engineering for Adaptive and Self-Managing Systems, Leipzig, Germany, pp. 65–72 (2008)
3. Brooks Jr., F.P.: The Design of Design: Essays from a Computer Scientist. Addison-Wesley, New York (2010)

4. 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.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009)
5. Buschmann, F., Henney, K., Schmidt, D.: Pattern-oriented software architecture: On patterns and pattern languages, vol. 5. John Wiley & Sons Inc. (2007)
6. 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, pp. 269–274 (2011)
7. Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A survey of autonomic communications. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 1, 223–259 (2006)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc. (1995)
9. 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.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 146–163. Springer, Heidelberg (2009)
10. Ghanbari, H., Litoiu, M.: Identifying implicitly declared self-tuning behavior through dynamic analysis. In: Proceedings of the 4th International Workshop on Software Engineering for Adaptive and Self-Managing Systems, Vancouver, BC, Canada, pp. 48–57 (2009)
11. Hellerstein, J., Diao, Y., Parekh, S., Tilbury, D.: Feedback control of computing systems, pp. 378–384. Wiley Interscience (2004)
12. IBM: An architectural blueprint for autonomic computing. (June 2006), http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf
13. Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., Carriere, J.: The architecture tradeoff analysis method. In: Proceedings of the 4th IEEE International Conference on Engineering of Complex Computer Systems, pp. 68–78 (1998)
14. Kephart, J., Chess, D.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
15. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: Future of Software Engineering, pp. 259–268 (2007)
16. Lane, T.G.: Studying software architecture through design spaces and rules. Tech. Rep. CMU/SEI-90-TR-18, Software Engineering Institute, Carnegie Mellon University (November 1990)
17. Litoiu, M.: Application performance evaluator and resource allocation tool (APERA) (May 2003), http://www.alphaworks.ibm.com/tech/apera
18. Litoiu, M., Woodside, M., Zheng, T.: Hierarchical model-based autonomic control of software systems. In: Proceedings of the Workshop on Design and Evolution of Autonomic Application Software, St. Louis, MO, USA, pp. 1–7 (2005)
19. Ramirez, A.J., Cheng, B.H.C.: Design patterns for developing dynamically adaptive systems. In: Proceedings of the 5th International Workshop on Software Engineering for Adaptive and Self-Managing Systems, Cape Town, South Africa, pp. 49–58 (2010)
20. Shaw, M.: The role of design spaces. IEEE Software (Special Issue on Studying Professional Software Design) 29(1), 46–50 (2012)
21. Smit, M.: Supporting Quality of Service, Configuration, and Autonomic Reconfiguration using Services-Aware Simulation. Ph.D. thesis, University of Alberta (2011)
22. Villegas, N.M., Müller, H.A., Tamura, G., Duchien, L., Casallas, R.: A framework for evaluating quality-driven self-adaptive software systems. In: Proceeding of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Waikiki, Honolulu, HI, USA, pp. 80–89 (2011)

Software Engineering Processes for Self-Adaptive Systems

Jesper Andersson[1], Luciano Baresi[2], Nelly Bencomo[3], Rogério de Lemos[4], Alessandra Gorla[5], Paola Inverardi[6], and Thomas Vogel[7]

[1] Department of Computer Science, Linnaeus University, Växjö, Sweden
[2] Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy
[3] INRIA Paris, Rocquencourt, France
[4] University of Kent, UK
[5] Faculty of Informatics, University of Lugano, Switzerland
[6] Dipartimento di Informatica, Università dell'Aquila, Italy
[7] Hasso Plattner Institute at the University of Potsdam, Germany

Abstract. In this paper, we discuss how for self-adaptive systems some activities that traditionally occur at development-time are moved to run-time. Responsibilities for these activities shift from software engineers to the system itself, causing the traditional boundary between development-time and run-time to blur. As a consequence, we argue how the traditional software engineering process needs to be reconceptualized to distinguish both development-time and run-time activities, and to support designers in taking decisions on how to properly engineer such systems.

Furthermore, we identify a number of challenges related to this required reconceptualization, and we propose initial ideas based on process modeling. We use the Software and Systems Process Engineering Meta-Model (SPEM) to specify which activities are meant to be performed off-line and on-line, and also the dependencies between them. The proposed models should capture information about the costs and benefits of shifting activities to run-time, since such models should support software engineers in their decisions when they are engineering self-adaptive systems.

1 Introduction

Traditional software engineering research primarily focuses on development activities for high-quality software, rather than maintenance or evolution [29]. Meanwhile, the software engineering community has accepted that software must continuously adapt and evolve according to ever changing requirements to remain useful for the user [26,27]. This awareness has led to iterative, incremental, and evolutionary software engineering processes [4,9,19,20,23,35,43], rather than strictly sequenced phases of requirements engineering, design, implementation, and testing, as perceived by the *waterfall model* [38].

However, such approaches to change software do not meet the requirements of many modern context and self-aware, mission-critical, or ultra-large-scale software systems [17,31]. Context and self-aware systems require *timely* changes in

R. de Lemos et al. (Eds.): Self-Adaptive Systems, LNCS 7475, pp. 51–75, 2013.

response to changing environments, changes in the system itself, or in its goals. The inherent delay in traditional change processes is for these systems unsatisfactory. Mission-critical systems have to operate continuously. In traditional change processes, changes are deployed during scheduled down-times and, as a consequence, continuous operation is not possible. Ultra-large-scale systems are highly complex, which makes human-driven change activities difficult and expensive, or even infeasible in practice due to the size and inherent complexity that impede a complete shutdown of the system in order to change it. Thus, we may conclude that using a traditional change process for these kinds of systems holds the risk of the system failing to meet its specification with respect to timely reaction to changes and continuous operation.

These risks have led to the development of novel means of risk mitigation that change software in terms of *self-adaptation* [14]. Self-adaptive behavior implies that certain development and change activities are shifted from development-time to run-time, while reassigning the responsibility for these activities from software engineers or administrators to the system itself. The new time-of-change timeline [11] that is, when a change takes place, covers development-time, deployment-time and run-time. A consequence of this reconceptualization of the time-of-change is that the traditional boundary between development-time and run-time blurs, which requires a complete reconceptualization of the software engineering process [3,7,21,22], where the traditional perspective that separates development-time and run-time is revisited. The rationale is that for self-adaptive software systems the typical software-process mapping from (1) *software life-cycle phases* [32] (e.g., development, deployment, operation, or evolution) and (2) *software engineering disciplines* [32] (e.g., requirements engineering, design, implementation, verification, etc.), and software-process activities (e.g., elicitation, prioritization and validation of requirements), onto a, (3) *time-of-change timeline* is not valid anymore. As an example, the disappearing boundary between development-time and run-time does not allow software changes to be decoupled from the running system anymore. Buckley et al. propose a taxonomy with a number of dimensions to classify software change [11]. Even though the proposed taxonomy appears to be sufficiently comprehensive, we argue that a more fine-grained perspective on the timeline is required for the class of self-adaptive software systems. This new perspective corresponds to a new dimension, which considers the blur and describes where, with respect to the self-adaptive system, change activities take place. With respect to the self-adaptive system, we refer to activities performed externally as *off-line activities* and to change activities performed internally as *on-line activities*[1].

The main contributions of this article is the identification and description of a number of challenges related to the required *reconceptualization of software engineering processes for self-adaptive systems*, and its impact on how to engineer such systems. We propose software process modeling as a corner-stone

[1] The notions of on-line and off-line we introduce are distinct from their traditional notions in the field of algorithm theory.

component to be used in a modeling and evaluation framework for self-adaptive software systems.

A reconceptualization of software engineering processes requires (1) *the definition of abstractions for off-line and on-line activities, the identification of the entities subject to change by these new activities and the dependencies in-between such activities* and, as a consequence, (2) *the complete understanding of the impact of design decisions related to off-line activities over on-line activities, and vice versa.* We argue that engineering self-adaptive software is about defining a software process which defines the scope for a system's self-adaptive behavior (by means of on-line activities) and proposes new variants to the mapping afore mentioned. However, scoping introduces a number of additional challenges.

We address these challenges with our second contribution, *process modeling for engineering self-adaptive software systems*. We propose an approach based on the *Software & Systems Process Engineering Meta-Model Specification* (SPEM) [32]. Explicit processes models manifest *how* a system is developed and evolved, which promotes a better understanding of self-adaptive software systems and the relationships to software engineering processes. In addition, it promotes communication, reuse, and reasoning that may in the long-term be automated [33]. Altogether this improves support for comprehension and decision making when engineering self-adaptive systems. The adoption of process modeling for self-adaptive software systems engineering involves a number of challenges: (1) The *new concepts* (on-line and off-line activities, and their dependencies) *have to be supported by the process modeling language.* (2) Likewise, *concepts and models must capture the relative value of activities (costs & benefits)* since they influence the scoping of a self-adaptive system's activities. Due to dependencies between the engineering process (off-line activities) and the self-adaptive system (on-line activities), we must develop and use (3) *new integrative design and modeling* techniques [6] that support both, process and product engineering. These techniques must support designers in scoping the self-adaptation mechanism, decisions that should be based on *value* and balance costs and benefits.

The remainder of this article is organized as follows. In Section 2, we introduce an illustrative example, which is used to define and motivate the problems this article is targeting. Section 3 discusses a number of challenges related to the problems in more depth. Based on these discussions, we outline a software process modeling approach for engineering self-adaptive software systems in Section 4. In Section 5, we discuss important challenges that remain to make process modeling a viable tool in self-adaptive software systems engineering. We discuss related work in Section 6, and conclude and outline a research agenda for the proposed direction in Section 7.

2 Revising the System Life-Cycle

In this section, we introduce a specific perspective concerning self-adaptive software systems, which is the characterization of their life-cycle and the challenges to support this life-cycle by effective software processes. With self-adaptive

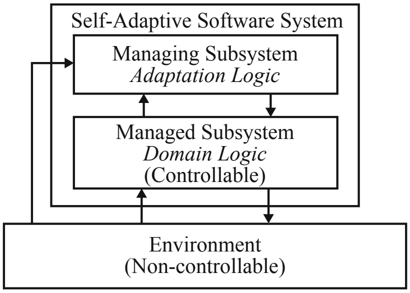


Fig. 1. A Conceptual Architecture for Self-Adaptive Software Systems [1]

software systems as our focal point, we revisit and revise the typical software system's life-cycle, pin-pointing explicit and implicit relationships between software engineering processes and a deployed self-adaptive software system.

The essential characteristic of a self-adaptive software system is its ability to autonomically evolve and adapt its behavior dynamically in response to changes to system requirements, the system itself, or the system's operational environment. The evolution and adaptation mechanisms should preserve the essence of the system behavior by continuously providing an acceptable implementation of the system's core requirements. The scope of this paper does not require a distinction between evolution and adaptation, the focus is whether *software changes* are enacted off-line or on-line and, as a consequence, a precise definition of software evolution and software adaptation is beyond the scope of this paper.

To frame evolution and adaptation mechanisms, Figure 1 depicts a conceptual architecture for self-adaptive software systems [1]. An important property of this architecture is the disciplined split, which promotes separation of concerns. A *Managing Subsystem* implements the *Adaptation Logic* that manages a *Managed Subsystem*, which implements the *Domain Logic*. The adaptation logic implements a control loop in line with the *monitor-analyze-plan-execute* (MAPE) loop [24], which evolves and adapts the domain logic. Domain functionality and the system's core requirements are implemented by the domain logic. The self-adaptive system operates in a non-controllable environment that may merely be observed by the adaptation logic, while the domain logic may both observe and affect the environment. Moreover, the architecture allows for additional managing subsystems that adapt adaptation logic in other managing subsystems. This may be used to describe, for example, the goal management layer in Kramer and Magee's layered architecture [25] or hierarchical control in autonomic computing [24]. This characterization of self-adaptive systems promotes evolution and adaptation mechanisms realized by the adaptation logic to first class computations that have to be supported side by side with domain logic computations, while seamlessly interleaving with the running system.

In a typical software process for conventional software systems, evolution and adaptation are performed after the initial development and deployment, and they usually encompass all process disciplines from requirements engineering to deployment. This indicates that timely changes are not prioritized. In a self-adaptive software system the situation is different. The evolution and adaptation activities realized by the adaptation logic are in place because requirements change frequently and timely reactions are essential. Self-adaptive software systems autonomically perform activities on-line and at run-time that originally have been carried out manually and off-line. This change to a process does not affect all process activities, and thus, understanding relationships and dependencies between (off-line) software process activities and (on-line) evolution and adaptation activities is a great challenge. Before systematically analyzing challenges concerning life-cycles and software processes for self-adaptive systems and the engineering of such systems, we present an illustrative example that we use throughout paper.

2.1 Illustrative Example: Automatic Workarounds

Automatic Workarounds (AW) is a technique that enhances applications with self-adaptation capabilities that deal with functional failures at run-time [12,13]. When applications fail, either because there is a fault in the application itself or in one of the libraries used by the application, the AW technique attempts to mask the fault and thus, to avoid the corresponding failures, while providing the core domain functionality.

The technique is based on the hypothesis that software systems usually offer several “equivalent operations” that provide the same core functionalities in different realizations. If an operation fails, the AW mechanism exploits this intrinsic redundancy to automatically find workarounds and apply an alternative equivalent sequence of operations. Consider for example a container component that implements one operation to add a single element, and one operation to add several elements. To add two elements, it is possible to add either one element after the other, or to add them both at the same time. If one of these options causes a failure at run-time, the AW technique attempts to execute the equivalent sequence of operations as the alternative option to mask the fault.

We depict the AW technique’s principal concepts and mechanisms in Figure 2. An application component invokes an operation provided by another component (*caller* and *called component*, respectively). If an invocation causes a failure, the AW technique implemented by the adaptation logic handles this failure at run-time by first looking for a sequence of operations which is equivalent to the failing invocation. Having found an equivalent sequence of operations, the adaptation logic enacts an adaptation that automatically invokes this sequence of operations. If the alternative execution does not cause a failure, a successful workaround has been found and the application proceeds as if the original failure never occurred. Otherwise, the adaptation logic continues testing other equivalent sequences until an alternative is either successfully executed or until all equivalent sequences have been tested unsuccessfully. In the latter case, the failure is reported to the caller component. To address these reported failures, developers have to fix the fault either by fixing the faulty component or by manually identifying and providing valid workarounds to the AW adaptation logic.

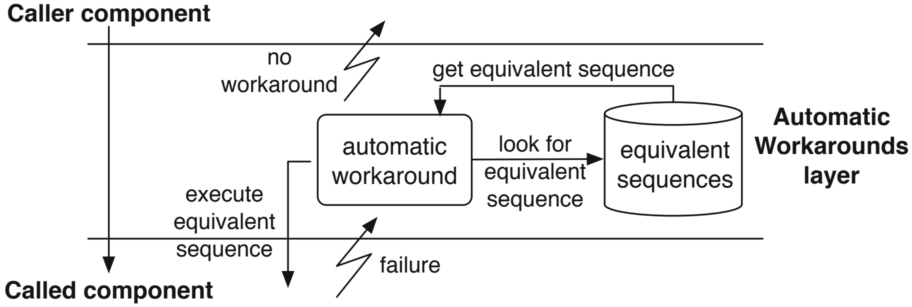


Fig. 2. The Automatic Workarounds technique

In a typical software process, the bug report filled by a user would be the starting point of a long manual effort to deal with run-time failures. Developers would need to identify and analyze the root causes of the problem, identify and implement a patch for the fault, and finally deploy a patch or re-deploy the complete application. The AW technique aims at automating and shifting these activities at run-time to the adaptation logic. This will provide for more timely responses to failures without the need to stop and re-deploy the application.

2.2 A Refined Life-Cycle Perspective

As mentioned above, a self-adaptive software system performs regular software process activities while the system is running. In Figure 3, we illustrate how a software process and its activities interact with a running self-adaptive software system. The left-hand side of the figure depicts a staged life-cycle model inspired by Rajlich and Bennett [35]. The stages cover the *initial development* of the self-adaptive system and traditional *evolution and adaptation* activities performed *off-line*. Off-line activities work on artifacts, such as design models or source code in a *product repository* and not directly on the running system. The final stage, *Phaseout* covers the shutdown and decommission of the self-adaptive system.

At first sight, and focusing on the left-hand side of the figure, this process looks identical to a traditional software process. However, note it interacts with the *running self-adaptive system*. This interaction takes place through *on-line* activities associated with *evolution and adaptation*, which constitute the self-adaptive system's adaptation logic. Using run-time representations of the self-adaptive system, on-line activities evolve and adapt the domain logic or other adaptation logic while the system is operational in providing services (illustrated by the right-hand side of Figure 3). Interactions and dependencies between off-line and on-line activities, depicted by bidirectional arrows, are specific for life-cycle models targeting self-adaptive systems.

In order to provide a more in-depth analysis of the interactions between a software process and a running self-adaptive system, more detailed descriptions of activities and their interactions are required. We introduce and discuss a timeline, illustrated in Figure 4, which represents a life-cycle instance for a

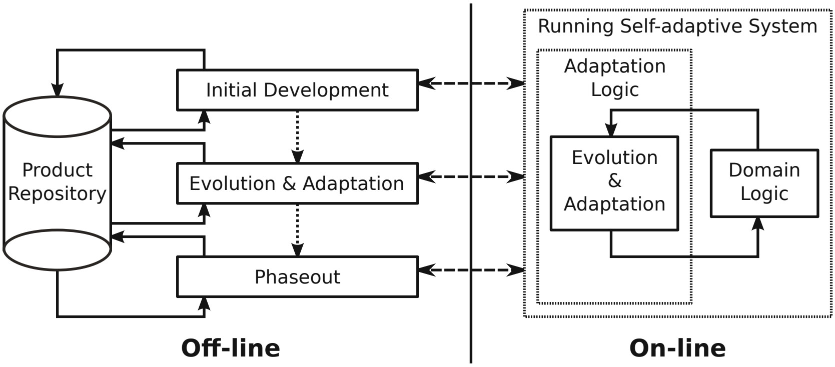


Fig. 3. A Life-cycle Model for Self-Adaptive Software System

self-adaptive system that uses an automatic workaround (AW) approach for the adaptation logic and its development process. The timeline view contains two graphs and *interaction points*. The top-most graph depicts the *activity level* (y-axis) in the development process and a number of specific *off-line activities* over *time* (x-axis). We see how the activity level varies over the life-cycle. The life-cycle is divided into the three distinct stages: initial development, evolution and adaptation, and phaseout. The bottom-most graph depicts the *service level* for the self-adaptive AW system over *time* (x-axis). The variations in the graph are due to events, external or internal to the system. For example, on-line activities initiated by the adaptation logic or the development processes (maintenance & evolution activities). The timeline in Figure 4 suggests that the running system acts as a stakeholder with a specific role in the development process, as it actively affects software development and maintenance [2]. However, for the case of self-adaptive systems, this is also true for on-line activities. We may now identify and characterize a number of scenarios where off-line and on-line activities interact, as depicted by the labeled *situations* in Figure 4.

The first stage, initial development, develops a first version of the self-adaptive software system by a number of off-line activities. In the context of the AW approach, software engineers develop the application's domain logic. To enhance the application with the AW technique, they have to provide an initial list of equivalent sequences for application operations with known workarounds, which is an off-line development activity. This also exemplifies how self-adaptation capabilities influence the initial development. Having completed the initial development, the system is ready for deployment. An initial deployment (cf. situation ① in Figure 4) puts the system into operation, which is illustrated by the step in the system's *service level*. The initial deployment activity is captured by the first *interaction point*. Interaction points indicate that off-line activities impact on-line activities or vice versa.

When the system instance is running, the evolution and adaptation stage starts. As we consider self-adaptive software systems, adaptations and evolution may be initiated and controlled by off-line (process) activities as well as

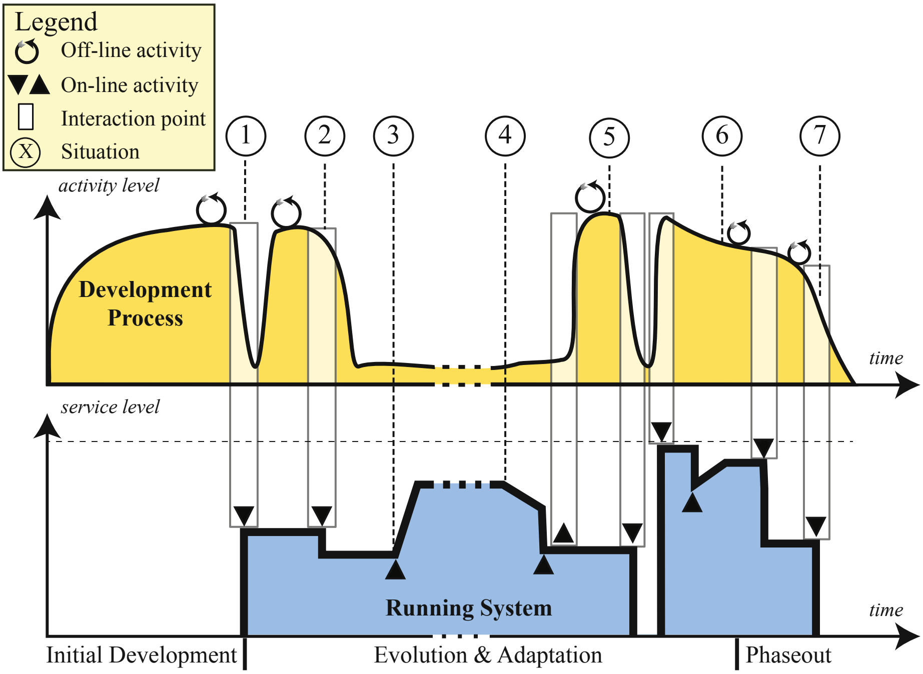


Fig. 4. Timeline View on a Process and a Running Self-Adaptive Software System

on-line (adaptation logic) activities. This is illustrated by six additional situations following initial deployment (situation (1)) in Figure 4.

Situation (2) illustrates how changes by off-line evolution or adaptation activities are subsequently enacted to the running system by on-line activities. An on-line update deploys new or updated domain logic components. For these components, the lists of equivalent sequences and thus, the adaptation logic, have to be updated by the developer to preserve AW behaviors. In the scenario we describe, the system's service level is affected negatively after the new deployment due to probable faults in the new components.

However, if corresponding failures occur, they are handled by the AW technique as sketched by situation (3), thanks to previous off-line efforts to maintain the list of equivalent sequences. The AW technique monitors failures in the application, and it is able to deal with them by successfully applying workarounds. This brings the system into a state with improved service level. This situation exemplifies on-line adaptations, which are often enabled by preceding off-line activities.

Nevertheless, the AW technique might not be able to cope with arbitrary failures that continuously affect the system's service level negatively (cf. situation (4)). This is the case when the AW technique does not find a valid workaround (first on-line activity in situation (4)), i.e., all available equivalent sequences have been tested without success, and as a consequence the failure recurs. In this case, the second on-line AW activity in this situation notifies developers who enact an off-line process to deal with the failures, e.g., by manually correcting the fault and maintaining the list of equivalent sequences. This situation shows how on-line activities interact with and trigger off-line activities.

If the application is re-engineered off-line, an on-line update may be too complex, hence not feasible. Such radical changes to a system are captured in situation ⑤ by off-line evolution activities followed by a deployment. In this situation the running system is shutdown and the new release deployed (similar to situation ①), which affects the system's availability and thus, its service level. Situation ⑥ highlights the case when off-line activities evolve or adapt on-line (adaptation logic) activities followed by enacting these changes to the running system. In context of the AW adaptation logic, at any point in time developers may identify and specify new equivalent sequences, which is an off-line activity that tunes the AW mechanism. Through on-line activities, these new sequences are injected into the AW knowledge and these sequences may be used in subsequent adaptations of the domain logic. Finally, situation ⑦ illustrates the complete shutdown and decommission as part of the phaseout stage, since a decision has been made to discontinue the system. The shutdown and decommission activities, which are planned and initiated off-line, terminate the life-cycle of the system and with some delay the process life-cycle.

Evolution and adaptation activities performed in-between interaction points are in general carried out on-line if they are controlled by the adaptation logic. In contrast, they are carried out off-line if they are controlled by human-driven process activities on product repository artifacts from the initial development stage (cf. Figure 3). In this context, interaction points synchronize off-line and on-line activities or artifacts. As an example, situation⑥ illustrates that on-line activities can be evolved and adapted off-line, and a subsequent dynamic update synchronizes these off-line changes to the corresponding on-line activities in the adaptation logic.

3 Processes for Self-Adaptive Software Systems

As we discussed previously, software processes for self-adaptive systems have special characteristics due to their integration with the running system by automating a set of process activities in the system's adaptation logic. This automation and integration define the self-adaptation scope, i.e., the self-adaptation capabilities of the system. In the case of the automatic workarounds approach example, this set covers activities that handle functional run-time failures, but not activities that go beyond the specific idea of workarounds, such as repairing the faults causing the failures (cf. Section 2.1). Therefore, we distinguish between *on-line* activities, which are change activities realized and performed by the system's adaptation logic, and *off-line* activities, which are realized and performed externally to the self-adaptive system (cf. Figure 3). Self-adaptation does not make typical (off-line) activities redundant. In fact, a process has to support both off-line and on-line activities. Furthermore, it has to consider dependencies in between both kinds of activities.

Currently, software processes merely focus on the initial development and off-line evolution and adaptation of the system. The automation and integration require a dramatic change of the concepts associated with traditional software processes. The software process reconceptualization is, therefore, the first of the major challenges we have identified. We believe that a process reconceptualization is essential to effectively and efficiently engineer self-adaptive software systems. We provide a more detailed account for this challenge and its subchallenges below in Section 3.1.

The outcome of the reconceptualization of software processes should lead to a life-cycle model that conveys a strong intertwining between the on-line and off-line activities and thus, between the self-adaptive system and its engineering process. We identify this as our second major challenge. Achieving effectiveness and efficiency in both is indeed a challenge that for instance includes deciding which activities should be performed on-line and which not. Design decisions of having activities either off-line or on-line should be motivated by the costs and the relative contribution to a product's *value*. In Section 3.2, we discuss related ideas and research challenges.

3.1 The Need of Reconceptualizing Software Processes

As mentioned above, traditionally software processes take the perspective that process activities can be either performed at development-time or at run-time. However, recently it has been advocated that due to the distinct features of self-adaptive systems, these type of systems require a reconceptualization of their software processes [3,7,21,22]. We already took the initial steps to modify the perspective on process activities in Section 2 by distinguishing between on-line and off-line activities rather than development-time and run-time activities.

Although on-line and off-line activities perform in different contexts, they are not independent from each other, as it is illustrated by the interactions in Figure 4. Therefore, an effective process for self-adaptive software systems should support both kinds of activities together with the interactions and dependencies.

In order to consider on-line activities, they must be lifted to the abstraction level of software processes. Up to now, on-line activities are only explicitly addressed by self-adaptive systems' designs that describe the adaptation logic. Thus, they are represented in software design models, but not in process models. Hence, to fully capture a software process for self-adaptive systems, on-line activities must be first class entities of processes, and they must be explicitly reflected in process models. It is therefore important to integrate the process models with the self-adaptive system design models in order to seamlessly capture off-line and on-line activities in a process.

Alongside on-line activities, on-line roles and work products need to be addressed by processes too. The adaptation logic of the self-adaptive system is responsible for performing on-line activities, since it assigns on-line process roles to the system. In a self-adaptive system, on-line activities will manipulate work products that are representations of the executing system. Thus, the process has to consider the running system, and more specifically the abstractions that reify adaptation and domain logic as work products of a process.

Having a side by side process support for on-line and off-line activities requires that work products and roles as well as dependencies between them are explicit. For instance, this is required to use dependencies to enable interactions between on-line and off-line activities or to synchronize on-line and off-line work products. Neglecting dependencies would split the process into two subprocesses that may drift apart, what could prevent controlled coevolution of the system and process.

3.2 Engineering Self-Adaptive Software System with Effective Process Support

The application of a reconceptualized process, in the way we have proposed above, requires support at the process level to effectively and efficiently engineer self-adaptive software systems. In fact, one of the consequences is that software engineers will have to face even more choices when engineering a self-adaptive system, as they have to decide on how to assign activities to off-line or on-line. Such engineering decisions have to be predictable and based on thorough value-based analysis of decision alternatives with well-understood consequences. Thus, we argue that decisions should be guided by models that represent both the costs and the benefits of having an activity performed either on-line or off-line together with the resulting dependencies. Such models should support engineers in making optimal decisions. In self-adaptive systems, some design decisions regarding the system's domain logic are delayed or revisited at run-time. The rationale for this delay is uncertainty, which in turn is a consequence of an information deficit [42]. For example, in the automatic workarounds approach, faults in the domain logic are not known at development-time, and thus, adaptation logic is integrated to cope with failures at run-time that are caused by such unknown faults. Thus, self-adaptation promotes shifting the tasks of traditional off-line activities, such as dealing with run-time failures, to on-line activities.

However, at the same time self-adaptation capabilities introduce additional uncertainty. When engineers leave design decision open to be resolved or revisited at run-time, it will be difficult to have all required information at hand that is required to make decisions with predictable consequences when designing a process and system. Thus, supporting the decision process and to deal with this type of uncertainty requires means to understand, specify, and reason about process activities, process roles, process work products, and dependencies. Such means must be provided at the process level, since any design decision concerning the self-adaptation scope affects the process, and any design decisions concerning the process influences the self-adaptation scope. Concretely, such decisions determine whether activities are performed on-line or off-line, and hence, whether they will become part of the adaptation logic or be handled off-line.

Moreover, such decisions must be based on the contribution to the product's value as perceived by its stakeholder using well-defined criteria that refer to activities, roles, work products, and dependencies. Such criteria specify costs and benefits for different design alternatives and support software engineers in making design decisions. For example in the automatic workarounds (AW) approach, using the AW technique to automatically deal with run-time failures induces additional costs for development (the equivalent sequences of operations must be specified by software developers), but it leads to the benefit of a more robust system, because the system is able to recover from run-time failures instead of crashing. The technique also introduces a performance penalty at run-time. Thus, a solid understanding of value, and useful ways to trade-off competing costs or benefits for the process and system together are required.

Besides issues concerning the design of self-adaptive systems and their processes, the supporting methods, techniques, and tools utilized by the activities are an additional parameter in the equation. Shifting activities from off-line to on-line requires that the underlying methods, techniques, and tools are adapted, optimized, or newly developed in order to be applicable on-line. This will result in a plethora of support alternatives for activities, and each alternative will have an associated value, possibly unique, for every given project and stakeholder. For example, an incremental solution to validation and verification can be efficient enough to be used on-line, but it might not provide the same degree of accuracy as a solution designed for off-line usage. Effective and efficient engineering requires that such value parameters are seamlessly integrated in the engineering process that codesigns on-line and off-line activities to define a process and the complementary adaptation logic in the managing system.

4 Process Modeling for Self-Adaptive Software Systems

In this section, we discuss a framework, based on software process modeling, that may help in engineering self-adaptive software systems. This requires that process modeling languages support the new concepts, like on-line and off-line activities, that originate from the reconceptualization discussed above. Thereby, process modeling also helps in grasping and understanding the reconceptualization of processes because process models make these concepts explicit. In addition, we discuss a number of remaining research challenges, primarily concerned with the development and application of a process modeling language for engineering self-adaptive software systems' processes.

In general, a *process* is a mechanism to achieve a goal in a systematic way, like following instructions to assemble a product, or following office procedures. Likewise, performing engineering activities to develop and evolve a software product constitutes a process [33]. How a process is carried out is specified by a *process model* that defines a partially ordered set of *what* is done *when* and *where* and by *whom* [16,36]. Thus, assembly instructions, office procedures, or descriptions of software engineering activities are process models, and such models materialize the corresponding processes. This materialization enables human understanding, coordination, and communication, and it supports the analysis, improvement, reuse, execution, or in general the management of processes [16,33,36].

To leverage such benefits of process modeling in the software engineering field, several modeling languages have been proposed to describe software processes [16]. One example of such modeling languages is the *Software & Systems Process Engineering Meta-Model Specification* (SPEM) [32]. In this paper, we

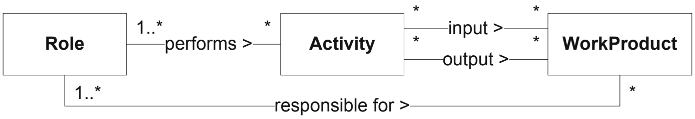


Fig. 5. A Conceptual View on the SPEM Meta-Model

have adopted SPEM for illustrating the concepts being discussed due to its flexibility, extensibility, and suitability for model-based engineering. The SPEM specification explicitly defines a modeling language by means of a meta-model for describing software development processes. Having an explicit definition of a modeling language, it is possible to discuss and extend the language. As discussed and demonstrated below, extending the language is required for addressing the concepts that originate from the reconceptualization of processes for the case of self-adaptive systems. We are not aware of any process modeling language that supports these concepts as first class elements and provides a rigorous underpinning for model-based engineering approaches.

4.1 SPEM-Based Process Modeling

By defining a meta-model, SPEM provides a modeling language to specify software development methods and processes, and offers an initial support for configuring and enacting processes in concrete projects. However, the language is generic, since it is meant to support the modeling of processes that span from the waterfall model to agile approaches. Thus, the language supports only the basic and abstract concepts that are present in any development approach. These abstract concepts are depicted in Figure 5, which shows a conceptual and partial view on the SPEM meta-model.

Using this meta-model, we are able to describe a workflow of *Activities* that are performed by *Roles* and that have *WorkProducts* as input or output. Activities can be related to other activities by passing work products along activities, i.e., an output of one activity is the input for another activity. Finally, responsibilities for work products can be assigned to roles. Moreover, the SPEM language provides several elements to represent phases, iterations, and milestones. Finally, SPEM allows to specify different forms of dependencies, e.g., between activities or between work products, primarily to cover relations between activities or composition and impact relations between work products. However, we do not further describe such advanced elements, as they are not critical for the specific challenges that this paper is addressing.

Below, we discuss how we have extended the basic concepts defined by the SPEM language with additional concepts that originate from the reconceptualization of software processes for the specific case of self-adaptive systems. This makes the additional concepts explicit in process models, and it helps to tackle challenges in engineering self-adaptive software systems (cf. Section 5). In the following section, we use our extended version of the SPEM language to model the process of the automatic workarounds approach, and we show how these

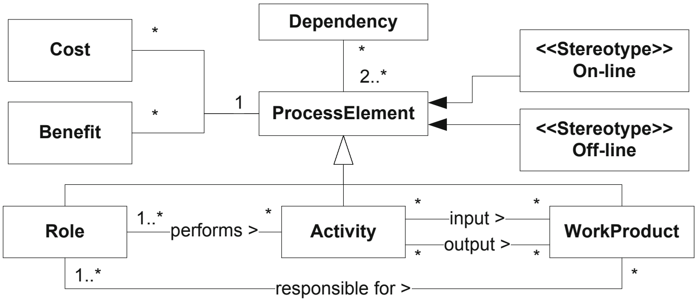


Fig. 6. Extended SPEM Meta-Model for Processes of Self-adaptive Systems

models can support and manifest the reconceptualization of a software process for self-adaptive systems.

4.2 Reconceptualization of SPEM-Based Process Modeling

As discussed above, the reconceptualization of software processes to address self-adaptive systems requires a new dimension for classifying activities. This dimension allows for a distinction between on-line and off-line situations, which, however, requires to make dependencies between on-line and off-line situations explicit and manageable. Moreover, costs and benefits of alternative on-line and off-line activities must be considered, as they guide the engineering and influence the designs of processes and systems. Therefore, as depicted in Figure 6, we extended the basic meta-model defined by SPEM with additional concepts. In the scope of this paper, these extensions, as well as the meta-models should be considered at the conceptual level, and not at the technical level as a definitive and fully specified modeling language. Thus, we do not discuss how the extensions could be best implemented or realized within the complete meta-model defined in the SPEM specification [32].

To keep the extensions to the original meta-model (cf. Figure 5) simple and generic, we have added the *ProcessElement* as a common super meta-class for the role, activity, and work product meta-classes. All further extensions refer to this *ProcessElement* and thus, they refer to all three concepts of role, activity, and work product. First of all, the *On-line* and *Off-line* stereotypes have been defined for process elements to clearly define whether any process element occurs on-line or off-line. More precisely, if the stereotype is associated with a role, it indicates whether the role is part of the self-adaptive system (on-line) or not (off-line). For a work product, it indicates whether such a work product is produced or generally used on-line or off-line. Likewise, applying the stereotypes to activities, process models may clearly distinguish whether any activity is performed on-line or off-line.

Moreover, we extended the SPEM modeling language with the concept of *Dependency* that relates two or more arbitrary process elements. This notion of dependency is more amenable and flexible for conceptual discussions than the specific possibilities provided by SPEM to cover, e.g., dependencies between activities or between work products. However, to implement these extensions within the SPEM meta-model, the already existing means to specify dependencies should be considered. Providing a generic notion of such concepts makes arbitrary dependencies, such as the different forms of interactions between on-line and off-line activities shown in Section 2, explicit in process models. As an example, being able to perform an activity on-line might require that another activity is in place off-line, which constitutes a dependency.

Finally, *Costs* and *Benefits* can be associated with any process element. This supports design decisions concerning the scoping of on-line and off-line activities, and should give answers to questions such as: “What are the costs and benefits of performing this activity on-line, in contrast to performing it off-line, and what other activities are affected or even required for the on-line and off-line variants?”

We now provide an example of how the extended SPEM language can be used to model the process of an application that relies on the AW approach for achieving self-adaptation, as described in Section 2.1. Figure 7 provides a high-level, structural view of the approach. This view includes all the core concepts of the SPEM language, which are the *Roles*, the *Activities*, and the *WorkProducts.*

Roles are depicted by actor icons, activities by rounded rectangles, and work products by document artifacts. As discussed above, these process elements can be stereotyped with *Off-line* or *On-line* to mark whether they belong to the off-line or on-line part of the process, respectively. Finally, dependencies are represented by rectangles connected to the interdependent process elements.

As shown in Figure 7, there are two roles in the AW approach: the *AW Layer* and the *Developer*. Both roles perform activities that have work products as input or output. The AW layer as the adaptation logic is part of the running self-adaptive system, and thus it is an on-line role. It monitors the execution of the application, i.e., the *Domain Logic*, on-line. If a *Failure* occurs, and it has been detected, the AW layer is in charge of either selecting a workaround, if any is known from previous executions, or searching for equivalent sequences, if no workaround is known. Thereby, the *Application Code with workarounds* is either selected or created by integrating the promising *Equivalent sequences* into the domain logic’s *Application code.* Finally, the AW layer enacts the adjusted application code and thus the adaptation by executing a known workaround or the equivalent sequences in an attempt to find a valid workaround.

The developer is an off-line role that maintains the list of equivalent sequences for the components of the domain logic. If the AW layer does not manage to find a valid workaround automatically, the developer fixes the related faults in the failing components and deploys the patched component to the running system. This requires that the maintained list of equivalent sequences for this component is updated in the AW layer for future on-line use. This update synchronizes the list of *Equivalent sequences* maintained off-line by the developer with the list of *Equivalent sequences* used on-line by the AW layer. This tackles the *To be synchronized* dependency between these two work products. Instead of hiding such dependencies in activities, they should be made explicit in the process models. Otherwise, they might get lost when the process and its activities change or evolve.

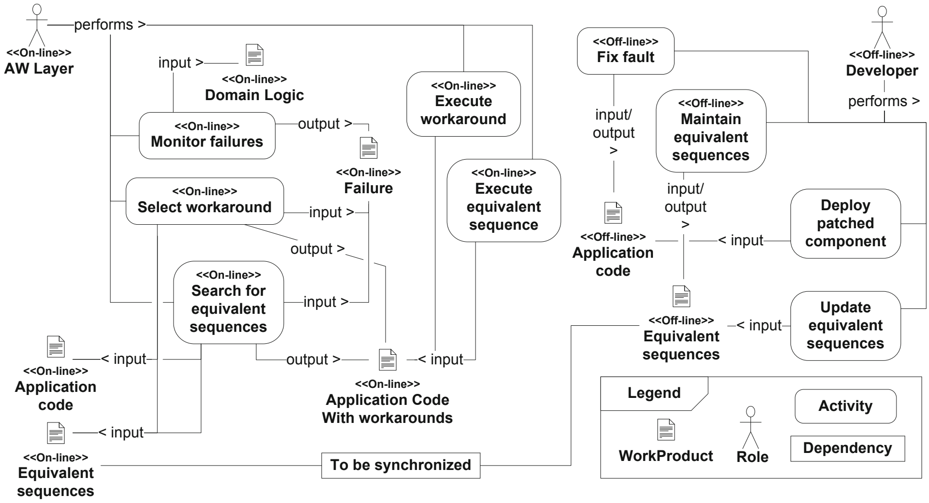


Fig. 7. *Roles*, *Activities*, and *WorkProducts* in the Automatic Workarounds approach

In addition to structural aspects, as depicted in Figure 7, the process behavior also needs to be specified. The original SPEM language allows to integrate external languages for behavior modeling, like *UML Activity* diagrams (cf. [32]). Likewise, the extended SPEM language we are proposing does not define its own behavior modeling formalism but uses UML activity diagrams. For our AW approach, Figure 8 depicts a UML activity diagram representing the workflow of activities for the case when a failure occurs and needs to be resolved.

The activity diagram represents the evolution and adaptation stage in the process timeline (cf. Figure 4) for a system that relies on the AW approach. The model consists of two partitions, one for each role, namely the AW layer and the developer. Each partition contains the activities performed by the corresponding role, and the model defines the workflow of activities within and across partitions respectively roles. The roles and activities used in the activity diagram are the same as in the structural process view depicted in Figure 7.

The AW layer monitors the status of the application to detect failures. When a failure occurs, it selects a workaround if any is already available from previous executions. If a workaround is available, the AW layer executes it immediately on-line. If no workaround is known, then the AW layer looks for equivalent sequences, and once it selects one, the selected sequence is executed. If the execution of the workaround or the equivalent sequence causes another failure, the loop continues until either one equivalent sequence does not cause any failure, or until there are no more equivalent sequences to try. In the last case, the developer has to fix the fault and maintain the list of equivalent sequences, which is followed by deploying the patched component and updating the list of equivalent sequences to make the off-line changes available to the AW layer in the running system.

In this section, we have shown that modeling processes for self-adaptive systems using an extended SPEM language lifts the concepts that originate from

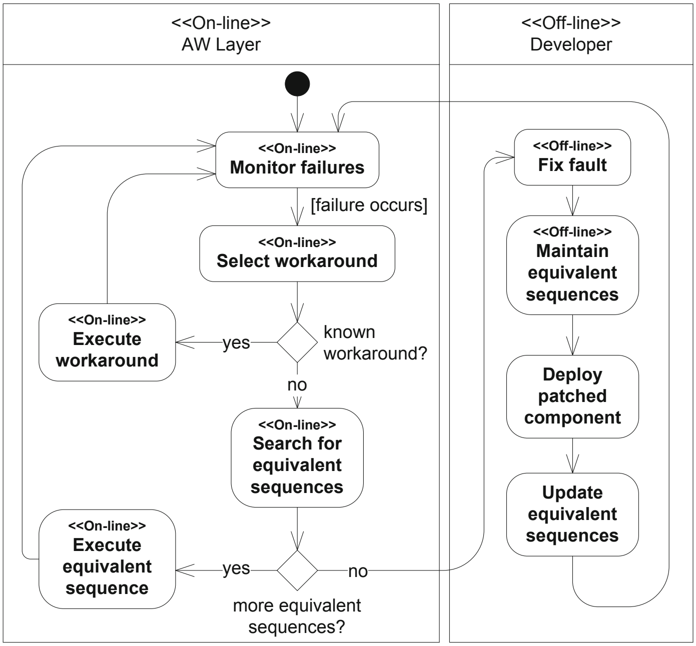


Fig. 8. Workflow of Activities in the Automatic Workarounds approach

the reconceptualized life-cycle to the abstraction level of software processes. This makes it possible to take a software process perspective on engineering self-adaptive systems, which is helpful in tackling the challenges related to the engineering of self-adaptive software systems.

5 Engineering Challenges

With the modeling aspect of the framework in place, we may shift focus to *design*, *decision making*, and *reasoning*. One of the key challenges we have identified in the engineering of self-adaptive systems is to partition the process activities effectively between off-line and on-line activities. As mentioned in the previous section, these activities may have costs and benefits associated to help in defining a process for each specific self-adaptive system that brings *value* to its stakeholders. *Value* here has a broad meaning that should encompass a number of aims: the system goals (quantitative and qualitative), the uncertainty that characterizes the execution environment (that defines the scope of adaptation) [42], the resource constraints of the execution environment (to support on-line activities), and the availability of accessing remote resources (to support off-line activities). This requires quantitative reasoning capabilities at the process definition level that shall suitably be complemented with stochastic reasoning to properly take into account the uncertainty dimension of the problem. As an illustration of the principles and practices in such reasoning support, we use *Value-Based Software Engineering* (VBSE) [6]. Biffl et al. argue that VBSE supports better software engineering decisions, providing an economic perspective where value bridges separation of concerns employed to manage complexity, thus allowing for achieving global optimums. The output of a software engineering process, the software system, has a number of goals associated. The purpose of an engineering process is to derive a solution, which optimizes the value (related to goals) of the product under current conditions. Engineering processes are characterized by their predictable outcomes, i.e., decisions made in a process have well-known consequences. Another characteristic is the continuous search for alternative solutions and an exhaustive evaluation of alternative solutions to provide sufficient knowledge on which decisions will be based.

The reconceptualization of software process activities, which allows some activities to migrate from off-line to on-line, dissolves a previously crisp boundary that separated software processes and the running system. This implies that the two may not be treated as separate concerns by an engineering process. Applying a value-based perspective on engineering a self-adaptive software system requires an understanding of value and that value is a main driver for the design and implementation of software processes for self-adaptive software systems and the systems themselves. Differently from how commonly intended in VBSE, cost in our context mainly concerns the impact of the activity in terms of resource consumption of the system's execution environment, e.g., computational time, memory use, etc. Benefit is the measure of the impact of the activity in terms of the system's goal, e.g., verification, graceful adaptation in presence of faults, etc. Value is the measure of the degree of satisfaction of the system's goals that can be achieved with the defined costs/benefits process tradeoffs.

VBSE is centered around Theory W [8] that aims at making all stakeholders in a project winners. VBSE suggests four supporting theories to "achieve and maintain a win-win state" [6, p. 19]; *dependency theory*, *utility theory*, *decision theory*, and *control theory*. Theory W and VBSE are developed with a process model which maintains watertight partitions between software processes and the running system. Our hypothesis is that the reconceptualization discussed above will impact VBSE in a fundamental way. However, the proposed approach where activities are modeled in a uniform way paves the way for customizing VBSE for developing self-adaptive software systems.

The first step in VBSE and Theory W is the identification of *success-critical stakeholders* (SCS). SCS are highly-important stakeholders and a project "will succeed if and only if it makes winners of *[the project's]* success-critical stakeholders" [6, p. 18]. It should be clear from the above discussion that in engineering a self-adaptive software system, the system itself is a SCS. Indeed, the extension proposed to the process modeling framework supports that the system is an SCS by making run-time roles, activities, and work-products explicit. At the same time, the execution environment characteristics, its resources and operational constraints, including its potential uncertain variability, represent another SCS. Depending on the self-adaptive system, the users of the system may represent another crucial SCS, they may, for example, define the acceptable behavioral variability of the system. The next step in VBSE identifies what is required to make a SCS a winner. Utility theory will play an important role here and may be used to define *value* on a level of detail where individual activities may define their win conditions. However, roles, activities, and work-products are not for free. We discussed the issue of relative cost above, and annotating process entities with costs will be essential in the next step where SCS should agree upon realization plans that will make all SCS winners. Eventually these plans will be implemented, a procedure which should be controlled to guarantee that the final products make all SCS winners.

The research challenge in this area is to formulate a VBSE theory and process for engineering self adaptive systems. We have described above some specific extensions to the SPEM meta-model that support value-based engineering. The *ProcessElement* in Figure 6 are annotated with collections of *Cost* and *Benefit* attributes. These concepts will represent the relative contribution of a specific *Role*, *Activity*, or *WorkProduct* to a *value*. The underlying idea with assigning costs and benefits to process elements is of course to use this knowledge in engineering activities. With the extended SPEM language, engineers are provided with the means to model and reason about how to design evolution and adaptation activities in a system. Distinguishing off-line activities and on-line activities opens up a design space where engineers may have several alternatives and eventually select the alternative that contributes relatively the most to the stakeholders' values. We illustrate some specific research challenges below using references from the automatic workaround example.

The first research challenge is to *provide the means for expressing value, that is the costs and benefits*. In the model we proposed, we describe off-line and on-line activities in an analogous way including cost and benefit attributes. This is an extremely simple approach to model *value*. We must develop ways for expressing *stakeholder specific values* in a way that they are useful for reasoning, evaluation, and eventually decision making. For example, the automatic workaround mechanism replaces a number of roles, activities, and work products. However it is not clear how to annotate these with costs, benefits, or any other type of value, neither for the process elements in the automatic workaround nor for its traditional, equivalent, off-line realization.

If we succeed in defining *a value-framework* for engineering self-adaptive software systems, engineers can reason about design alternatives, evaluate, and make predictable decisions about the relative contribution to the overall value. However, it is not clear how to reason about and evaluate alternatives in a structured manner. This takes us to the second challenge, *we need to design new reasoning and evaluation techniques*, potentially based on the large body of existing value-based design methods, for instance [15]. The design of a system's adaptation subsystem will require that engineers decide if and which activity should be performed off-line or on-line. Consider for example the *Maintain equivalent sequences* activity in Figure 7. It is performed by a *Developer*. However, it is not unlikely that future evolution of the AW mechanism provides for additional alternative realizations of this activity, for instance, by means of other automatic on-line activities. In that situation, engineers have a selection of alternatives to choose from in order to select the combination with the greatest relative contribution to the stakeholders' values.

Another challenge associated with processes for self-adaptive software systems is the fact that processes need to be generated dynamically at run-time since changes affecting the system, its context and goals may require their adaptation. This may imply that depending on the system's operational conditions, different processes can be generated by changing their activities or workflows. Moreover, since off-line and on-line activities might influence each other, it is important to consider how the initial development-time design rationale can affect the processes being generated at run-time, and vice versa. Finally, it is also crucial to incorporate into off-line activities the decisions being made during run-time since they would provide insightful knowledge about the operational profile of the system.

6 Related Work

Different researchers like Finkelstein and Blair et al. [7] or Inverardi and Tivoli [21,22] have also identified the need for new software engineering paradigms suggesting a reconceptualization of software processes. Among others, this is motivated by the blurring boundary between development-time and run-time as discussed in [3,7]. This is inline with the motivation for our work in this paper on revising the life-cycle and processes for self-adaptive software systems.

Challenges for software evolution are also discussed by Mens et al. who specifically state that "*[I]*t is important to investigate how the notion of software change can be integrated into the conventional software development process models" [30, p. 17]. They consider agile or in general iterative and incremental development processes as promising approaches to integrate support for change in the life-cycle. In contrast to this paper, they do not focus on life-cycle or process issues related to changes by means of self-adaptation or related to the blurring boundary between development-time and run-time. Likewise, Buckley et al. [11] or McKinley et al. [28] clearly distinguish between changes performed statically at development-time or dynamically at run-time. This is based on a traditional view on a system life-cycle, while we promote a refined view that primarily considers on-line and off-line changes or in general on-line and off-line process activities. In this context, by on-line and off-line we refer to different ways changes are carried out, but not whether the running system's domain logic provides service or not while being changed (cf. *availability* dimension in [11]).

Salehie and Tahvildari [39] discuss research challenges for the specific case of self-adaptive software, but not from a process view. They briefly consider a developing phase and an operating phase for self-adaptive systems, while the developing phase determines the adaptation capabilities in the operational phase. However, these phases are not used for discussing the challenges and in particular, this distinction into these two phases is similar to the traditional view on life-cycles exclusively separating development-time and run-time.

Gacek et al. [18] view evolution and adaptation as processes that include roles, artifacts etc., which is similar to our work. They propose a self-adaptation reference process that consists of two iteratively interacting processes. The inner adaptation process addresses the component control and change management layers of the reference architecture by Kramer and Magee [25], while the outer evolution process relates to the goal management layer. The authors conceptually discuss how a manual or partially automated evolution process guides an automated adaptation process by interactions between these two processes. These interactions seem to be similar to the interaction points between on-line and off-line process activities that we discussed in the context of Figure 4. However, Gacek et al. focus on discussing the co-existence of self-adaptation and traditional change management or evolution, but they do not discuss implications on software engineering processes or system life-cycles as this paper does.

Other approaches investigating software processes for self-* systems and especially for adaptive multi-agent systems cover only the development of such systems and not the whole life-cycle [34,37]. This means that the adaptation mechanisms are exclusively considered as part of the system to be developed, but not as a part of the life-cycle process itself. Thus, both approaches [34,37] describe processes or methodologies that specify the development of the systems including the development of the adaptation mechanisms. In contrast, besides considering adaptation mechanisms as part of a self-adaptive system, we also lift the adaptation mechanisms to the process level by treating the adaptation logic as a process role and the tasks performed by the adaptation logic as process activities. Consequently, we address processes that describe the whole system life-cycle comprising the development as well as the adaptation and evolution of the system. Nevertheless, one commonality between our work and [34,37] is the usage of the same process modeling language, namely SPEM, though we conceptually extended SPEM due to the required reconceptualization of software processes.

Our work is also motivated by the fact that approaches to modern or self-adaptive software systems do not design comprehensive software processes spanning on-line and off-line activities for their approaches, and they just shift specific typical process activities to the system in order to be performed on-line. For example, Brenner et al. [10] equips components with mechanisms to test them on-line, which shifts typical validation and verification activities and efforts to the run-time. Another example is the work of Bencomo et al. [5] who consider a self-adaptive system as a dynamic software product line that determines product configurations on-line and at run-time, while for traditional product lines the configurations are determined off-line and usually before deployment. Such approaches can benefit from our work since we provide initial means to model and analyze processes that cover both on-line and off-line activities. This might help other approaches to engineer their systems with effective process support.

Another initiative associated with processes for self-adaptive software systems is the dynamic generation of plans at run-time [40,41]. A key factor motivating this work was how to deal with the uncertainty related to changing goals, unexpected resource conditions, and unpredictable environments when managing the adaptation of software systems. This has shown particularly relevance when applied to the generation of plans for managing the integration testing of self-adaptive systems [41], which is a process that involves to calculate integration order, generate stubs and test cases, and perform the actual tests. Although this work is restricted to on-line activities, it would be interesting to consider how off-line activities could affect the automatic generation of plans, and how cost and benefit could be integrated with the decision making of selecting the most appropriate plan.

7 Conclusion and Future Work

The actual support for self-adaptation throughout the entire life-cycle of self-adaptive software systems requires a reconceptualization of the way they are engineered. Therefore, we presented a first integrated view of the problem, suitable abstractions for off-line and on-line process activities, and details of major challenges concerning the reconceptualization of software processes for self-adaptive software systems. Moreover, for tackling the challenges related to the engineering of self-adaptive software systems, we proposed an approach based on process modeling and value-based software engineering. An essential part of this approach is the intertwining of a self-adaptive software system and its software process.

As future work, we plan to elaborate the reconceptualization of software processes for self-adaptive systems, e.g., by investigating the impact of the on-line/off-line perspective on state-of-the-art approaches, methods, and techniques to design software processes and to engineer self-adaptive systems. Having more profound knowledge about reconceptualized software processes, we can work on formalizing the modeling language to fully capture a process and its system. A formal language is the prerequisite for automated analysis that follows the theory of value-based software engineering. Therefore, we have to adapt this theory to address specifics of self-adaptive systems and processes for such systems. For instance, we need to think about benchmarks and special-purpose metrics to assess processes and the corresponding self-adaptive systems as well as their values based on costs and benefits. How can we say that a given process is better than another one at identifying, designing, implementing, and running the on-line and off-line process activities for a system?

Besides these initial directions, a possible research agenda should in particular comprise the following elements. We need to better understand how to elicit functional and non-functional requirements of self-adaptive systems, especially, on how these should be associated with on-line and off-line activities and their dependencies. This could lead to a model-driven solution for the development, deployment, adaptation, and evolution of these system. We also need to better understand the dependencies between self-adaptation and software evolution since the former does not imply replacing the latter. This requires clear definitions of (self-)adaptation and evolution, and how both can be seamlessly integrated in a self-adaptive system's process. For example, there might be the need for understanding how to evolve the system based on its run-time adaptations. Experiences from the adaptations performed in the past may offer useful knowledge for the evolution of the system.

All these research directions promote our ultimate goal of effectively and efficiently engineering self-adaptive systems with proper software process support. Thereby, the process perspective should leverage systematic approaches to engineering self-adaptive systems, which have predictable outcomes concerning effectiveness and efficiency.

Acknowledgment. This paper is the result of stimulating discussions among the authors and other participants, especially Bojan Cukic, Oscar M. Nierstrasz, Sooyong Park, and Dennis B. Smith, during the seminar on *Software Engineering for Self-Adaptive Systems* at Schloss Dagstuhl in October 2010 (`http://www.dagstuhl.de/10431`).

References

1. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Reflecting on self-adaptive software systems. In: Proc. of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2009), pp. 38–47. IEEE Computer Society (2009)
2. Bai, X., Huang, L., Zhang, H.: On Scoping Stakeholders and Artifacts in Software Process. In: Münch, J., Yang, Y., Schäfer, W. (eds.) ICSP 2010. LNCS, vol. 6195, pp. 39–51. Springer, Heidelberg (2010)
3. Baresi, L., Ghezzi, C.: The disappearing boundary between development-time and run-time. In: Proc. of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER 2010), pp. 17–22. ACM, New York (2010)
4. Beck, K.: Embracing Change with Extreme Programming. IEEE Computer 32(10), 70–77 (1999)
5. Bencomo, N., Sawyer, P., Blair, G., Grace, P.: Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In: Thiel, S., Pohl, K. (eds.) Proc. of the 12th International Software Product Line Conference (SPLC 2008), Second Volume (Workshops), pp. 23–32. Lero Int. Science Centre, University of Limerick, Ireland (2008)
6. Biffl, S., Aurum, A., Boehm, B., Erdogmus, H., Grünbacher, P. (eds.): Value-Based Software Engineering. Springer (2006)
7. Blair, G., Bencomo, N., France, R.B.: Models@run.time: Guest Editors' Introduction. IEEE Computer 42(10), 22–27 (2009)
8. Boehm, B.W., Ross, R.: Theory-W Software Project Management Principles and Examples. IEEE Trans. Softw. Eng. 15(7), 902–916 (1989)
9. Boehm, B.W.: A Spiral Model of Software Development and Enhancement. IEEE Computer 21(5), 61–72 (1988)
10. Brenner, D., Atkinson, C., Malaka, R., Merdes, M., Paech, B., Suliman, D.: Reducing verification effort in component-based software engineering through built-in testing. Information Systems Frontiers 9(2), 151–162 (2007)
11. Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G.: Towards a taxonomy of software change. Journal of Software Maintenance and Evolution: Research and Practice 17(5), 309–332 (2005)

12. Carzaniga, A., Gorla, A., Perino, N., Pezzè, M.: Automatic workarounds for web applications. In: Proc. of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2010), pp. 237–246. ACM, New York (2010)
13. Carzaniga, A., Gorla, A., Pezzè, M.: Self-healing by means of automatic workarounds. In: Proc. of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008), pp. 17–24. ACM, New York (2008)
14. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: 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)
15. Clements, P., Kazman, R., Klein, M.: Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley, Boston (2001)
16. Curtis, B., Kellner, M.I., Over, J.: Process modeling. Commun. ACM 35(9), 75–90 (1992)
17. Gabriel, R.P., Northrop, L., Schmidt, D.C., Sullivan, K.: Ultra-large-scale systems. In: OOPSLA 2006: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, pp. 632–634. ACM, New York (2006)
18. Gacek, C., Giese, H., Hadar, E.: Friends or foes?: a conceptual analysis of self-adaptation and it change management. In: Proc. of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008), pp. 121–128. ACM, New York (2008)
19. Gilb, T.: Evolutionary development. SIGSOFT Softw. Eng. Notes 6(2), 17 (1981)
20. Gilb, T.: Evolutionary Delivery versus the waterfall model. SIGSOFT Softw. Eng. Notes 10(3), 49–61 (1985)
21. 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)
22. Inverardi, P., Tivoli, M.: The Future of Software: Adaptation and Dependability. In: De Lucia, A., Ferrucci, F. (eds.) ISSSE 2006-2008. LNCS, vol. 5413, pp. 1–31. Springer, Heidelberg (2009)
23. Jacobson, I., Booch, G., Rumbaugh, J.: The unified process. IEEE Software 16(3), 96–102 (1999)
24. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
25. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: Future of Software Engineering (FOSE 2007), pp. 259–268. IEEE Computer Society (2007)
26. Lehman, M.M.: Software's Future: Managing Evolution. IEEE Software 15(01), 40–44 (1998)
27. Lehman, M.M., Belady, L.A. (eds.): Program evolution: processes of software change. Academic Press Professional, Inc., San Diego (1985)
28. McKinley, P., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing Adaptive Software. IEEE Computer 37(7), 56–64 (2004)
29. Mens, T.: Introduction and Roadmap: History and Challenges of Software Evolution. In: Software Evolution, ch.1. Springer (2008)

30. Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M.: Challenges in software evolution. In: Proc. of the 8th International Workshop on Principles of Software Evolution (IWPSE 2005), pp. 13–22. IEEE Computer Society (2005)
31. Northrop, L., Feiler, P.H., Gabriel, R.P., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D.: Ultra-Large-Scale Systems: The Software Challenge of the Future. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2006)
32. Object Management Group (OMG): Software & Systems Process Engineering Meta-Model Specification (SPEM), Version 2.0 (2008)
33. Osterweil, L.J.: Software processes are software too. In: Proc. of the 9th International Conference on Software Engineering (ICSE 1987), pp. 2–13. IEEE Computer Society, Los Alamitos (1987)
34. Puviani, M., Serugendo, G.D.M., Frei, R., Cabri, G.: Methodologies for self-organising systems: A spem approach. In: Proc. of the IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT 2009), vol. 02, pp. 66–69. IEEE Computer Society (2009)
35. Rajlich, V.T., Bennett, K.H.: A Staged Model for the Software Life Cycle. IEEE Computer 33(7), 66–71 (2000)
36. Rolland, C.: Modeling the requirements engineering process. In: Markus, A.F., Jaakkola, H., Tadahiro, K., Kangassalo, H. (eds.) Information Modelling and Knowledge Bases V: Principles and Formal Techniques: Results of the 3rd European-Japanese Seminar, Budapest, Hungary, May 31-June 3, pp. 85–96. IOS Press (1994)
37. Rougemaille, S., Migeon, F., Millan, T., Gleizes, M.-P.: Methodology Fragments Definition in SPEM for Designing Adaptive Methodology: A First Step. In: Luck, M., Gomez-Sanz, J.J. (eds.) AOSE 2008. LNCS, vol. 5386, pp. 74–85. Springer, Heidelberg (2009)
38. Royce, W.: Managing the Development of Large Software Systems: Concepts and Techniques. In: Proc. IEEE WESTCON. IEEE Computer Society Press (1970); Reprinted in Proc. of the 9th International Conference on Software Engineering (ICSE 1987), pp. 328-338. IEEE Computer Society
39. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2), 1–42 (2009)
40. da Silva, C.E., de Lemos, R.: Using dynamic workflows for coordinating self-adaptation of software systems. In: Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2009), pp. 86–95. IEEE Computer Society, Washington, DC (2009)
41. da Silva, C.E., de Lemos, R.: Dynamic plans for integration testing of self-adaptive software systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011), pp. 148–157. ACM, New York (2011)
42. Welsh, K., Sawyer, P.: Understanding the Scope of Uncertainty in Dynamically Adaptive Systems. In: Wieringa, R., Persson, A. (eds.) REFSQ 2010. LNCS, vol. 6182, pp. 2–16. Springer, Heidelberg (2010)
43. Yau, S.S., Colofello, J.S., MacGregor, T.: Ripple effect analysis of software maintenance. In: Proc. of the 2nd International Conference on Computer Software and Applications (COMPSAC 1978), pp. 60–65. IEEE Computer Society (1978)

On Patterns for Decentralized Control in Self-Adaptive Systems

Danny Weyns[1], Bradley Schmerl[2], Vincenzo Grassi[3], Sam Malek[4],
Raffaela Mirandola[5], Christian Prehofer[6], Jochen Wuttke[7],
Jesper Andersson[1], Holger Giese[8], and Karl Göschka[9]

[1] Linnaeus University, Växjö, Sweden
[2] Carnegie Mellon University, Pittsburgh, PA, USA
[3] Università di Roma TorVergata, Italy
[4] George Mason University, Washington DC, USA
[5] Politecnico di Milano, Milan, Italy
[6] LMU München and Fraunhofer ESK, Germany
[7] University of Washington, WA, USA
[8] Hasso Plattner Institute at the University of Potsdam, Germany
[9] Technische Universität Wien, Austria

Abstract. Self-adaptation is typically realized using a control loop. One prominent approach for organizing a control loop in self-adaptive systems is by means of four components that are responsible for the primary functions of self-adaptation: Monitor, Analyze, Plan, and Execute, together forming a MAPE loop. When systems are large, complex, and heterogeneous, a single MAPE loop may not be sufficient for managing all adaptation in a system, so multiple MAPE loops may be introduced. In self-adaptive systems with multiple MAPE loops, decisions about how to decentralize each of the MAPE functions must be made. These decisions involve how and whether the corresponding functions from multiple loops are to be coordinated (e.g., planning components coordinating to prepare a plan for an adaptation). To foster comprehension of self-adaptive systems with multiple MAPE loops and support reuse of known solutions, it is crucial that we document common design approaches for engineers. As such systematic knowledge is currently lacking, it is timely to reflect on these systems to: (a) consolidate the knowledge in this area, and (b) to develop a systematic approach for describing different types of control in self-adaptive systems. We contribute with a simple notation for describing interacting MAPE loops, which we believe helps in achieving (b), and we use this notation to describe a number of existing patterns of interacting MAPE loops, to begin to fulfill (a). From our study, we outline numerous remaining research challenges in this area.

1 Introduction

Self-adaptive systems have the ability to adapt themselves to changes in their execution environment and internal dynamics, such as response to failure, variability in available resources, or changing user priorities, to continue to achieve their goals.

R. de Lemos et al. (Eds.): Self-Adaptive Systems, LNCS 7475, pp. 76–107, 2013.

Examples of self-adaptive systems are those that optimize their performance under changing operating conditions, and systems that heal themselves when certain components fail. Feedback control loops have been identified as crucial elements in realizing self-adaptation of software systems [46,28,11]. One prominent approach to organizing a control loop in self-adaptive sytems is by means of four components that are responsible for the primary functions of self-adaptation: Monitor, Analyze, Plan, and Execute, often referred to as the MAPE loop [28]. When systems are large, complex, and heterogeneous, a single MAPE loop may not be sufficient for managing adaptation [9,1]. In such cases, multiple MAPE loops may be employed that manage different parts of the system. In self-adaptive systems with multiple MAPE loops, the functions for monitoring, analyzing, planning, and effecting may be made by multiple components that coordinate with one another. That is, the functions may be *decentralized* throughout the multiple MAPE loops. An example is a self-adaptive system in which multiple planning components coordinate with one another to prepare a plan for adaptation.

Different patterns of interacting control loops have been used in practice by centralizing and decentralizing the functions of self-adaption in different ways. For example, in the Rainbow framework [17], monitoring and execution are delegated to the different nodes of the controlled system, whereas analyzing and planning are centralized. The IBM architectural blueprint [25] organizes MAPE loops hierarchically, where each level of the hierarchy contains instances of all four MAPE components. In this setting, higher level MAPE loops determine the set values for the subordinate MAPE loops. In fully decentralized settings, relatively independent MAPE components coordinate with one another and adapt the system when needed. An example of this approach is discussed in [18], in which local component managers on different nodes coordinate with one another to (re-)configure the structure of the managed system according to the overall architectural specification.

The existing self-adaptive literature and research, in particular those with a software engineering perspective, have by and large tackled the problem of managing either local or distributed software systems in a centralized or hierarchical fashion, e.g., [40,17,25]. While increasing attention is given to decentralized control of self-adaptive software, e.g., [12,18,36,5,56,48,53], we believe that there is a dearth of practical and effective techniques to build systems in this fashion. However, there is an opportunity to build on the work of decentralized self-adaptation to understand the recurrent coordination patterns and trade-offs, so that systematic design of self-adaptive systems can be acheived.

To foster comprehension of self-adaptive systems with multiple control loops, and support reuse of known solutions in this area, it is crucial that we document common design approaches for engineers [47]. However, systematic knowledge about interacting control loops for self-adaptive systems is lacking. Therefore, it is timely to reflect on these systems to: (a) consolidate the knowledge in this area, and (b) develop a systematic approach for describing different types of control in self-adaptive systems. In this chapter, we contribute with a simple notation for describing multiple interacting MAPE loops, which we believe helps in achieving (b), and we use this notation to describe a number of well-known patterns of interacting MAPE loops, to begin to fulfill (a). Patterns are an established approach for documenting systematic knowledge in a particular area. A pattern describes a generic solution for a recurring design problem. The patterns we present are derived from common knowledge in the field of self-adaptation and experiences of the authors with building self-adaptive systems. In reflecting about these patterns and the different ways of organizing self-adaptive control loops, we have identified a number of further research challenges that together form a roadmap for achieving a more principled approach to designing decentralized self-adaptive systems. This roadmap is outlined in the conclusion of this chapter.

2 Terminology

Before we elaborate on the notation for interacting MAPE loops and the patterns, we first clarify terminology. In particular, we (1) explain the distinction between managed and managing subsystems, the two constituent parts of a self-adaptive system, and (2) clarify how we use the terms distribution and decentralization in this chapter, two terms that are often mixed up by software engineers in the community of self-adaptive systems, leading to a lot of confusion.

2.1 Managed and Managing Subsystem

As shown in Figure 1, a self-adaptive system is situated in an environment. We use the general terms *managed* subsystem and *managing* subsystem to denote the constituent parts of a self-adaptive software system. Other authors make a similar distinction. For example, in the Rainbow framework [17], the managed subsystem maps to the system layer and the managing subsystem to the architecture layer. The authors in [43] use core function to refer to the managed subsystem and adaptation engine to refer to the managing subsystem. In FORMS [57], the managed subsystem corresponds to the base-level subsystem, and the managing subsystem to the reflective subsystem.

The environment refers to the part of the external world with which the self-adaptive system interacts, and in which the effects of the system will be observed and evaluated [26]. The environment may correspond to both physical and software entities. For example, the environment of a robotic system includes physical entities like obstacles on the robot's path and other robots, as well as external cameras and corresponding software drivers. The distinction between the environment and the self-adaptive system is made based on the extent of control. For instance, in the robotic system, the self-adaptive system may interface with the mountable camera sensor, but since it does not manage (adapt) its functionality, the camera is considered to be part of the environment.

The managed subsystem comprises the application logic that provides the system's domain functionality. For instance, in the case of robots, navigation of a robot or transporting loads is performed by the managed subsystem. To realize its functionality, the managed subsystem monitors and affects the environment.

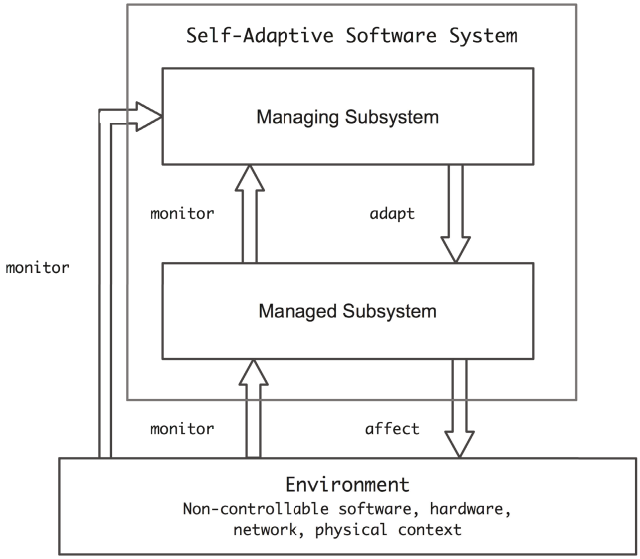

Self-Adaptive Software System
Managing Subsystem
monitor
adapt
monitor
Managed Subsystem
monitor
affect
Environment
Non-controllable software, hardware, network, physical context

Fig. 1. Constituent parts of a self-adaptive software system

To support adaptations, the managed subsystem has to provide support for monitoring and executing adaptations.

The managing subsystem manages the managed subsystem. The managing subsystem comprises the adaptation logic that deals with one or more concerns. For instance, a robot may be equipped with a managing subsystem that allows adaption of its navigation strategy based on the changing operation conditions, e.g., changing task load, or reduced bandwidth for communication. To realize its goals, the managing subsystem monitors the environment and the managed subsystem and adapts the latter when necessary.

Other layers can be added to the system where higher-level managing subsystems manage underlying subsystems, which can be managing subsystems themselves. For instance, consider a robot that not only has the ability to adapt its navigation strategy, but also to adapt the way such adaptation decisions are made, e.g., based on remaining energy level of the battery. In such an instance, the subsystem responsible for managing the battery level of the robot must coordinate with the subsystem for managing navigation and other robotics tasks, so that the robot does not fail entirely.

It is important to note that the managed and managing subsystems can be interwoven, as is the case when adaptation logic is dispersed throughout the functional logic the system. In such systems, it is not possible to easily reason about adaptation logic separately from system logic, meaning it is difficult to provide assurances or guarantees on the behavior of the system to changes in the environment. Another emerging approach to self-adaptation is in the field of self-organizing systems, where adaptation comes entirely from decisions made locally by components of the system. In such systems, the global properties of the adaptation (e.g., performance, utility to the user, or failure properties) are also difficult to reason about, though there is some research that attempts to address these concerns (for example, in [51], the authors present a statistical model that allows the convergence of global system objectives based on local agent behaviors to be analyzed, predicted, and controlled). In this chapter, we focus on how to organize self-adaptive systems where both subsystems are separate entities, following the principle of disciplined split [34] (or separation of concerns), which has been a main focus in the self-adaptive research community [55].

2.2 Distribution and Decentralization

Textbooks on distributed systems, e.g., [49], typically differentiate between: (1) centralized data in contrast to distributed, partitioned, and replicated data, (2) centralized services in contrast to distributed, partitioned, and replicated services, and (3) centralized algorithms in contrast to decentralized algorithms.

In this chapter, we use *distribution* to refer to the deployment of the software of a self-adaptive system to hardware. As such, distribution of a self-adaptive system refers to the deployment of the software of both the managed subsystem and the managing subsystem. A distributed self-adaptive system consists of multiple software components that are deployed on multiple nodes connected via some network. The opposite of a distributed self-adaptive system is software that is deployed on a single node. The managed and managing subsystems can be deployed on the same or on different nodes. For example, the software components of a managed subsystem may be deployed on a set of nodes, while the software of the managing system may be deployed on one dedicated node. Thus, while the managed system may be distributed, it is possible that the managing system is not.

With *decentralization*, we refer to how control decisions in a self-adaptive software system are coordinated among different components, independent of how those control components are physically distributed. In particular, we consider decentralization at the level of the four activities of self-adaption: monitoring, analyzing, planning, and execution. Decentralization implies a type of control in which multiple components responsible for one of the activities of self-adaption perform their functionality locally, but coordinated with with peers. Typically, such decentralized coordination is organized as follows: monitoring components coordinate with other monitoring components to collect the knowledge required for subsequent analysis; analysis components coordinate to decide whether the conditions for a particular adaptation hold; multiple planning components coordinate to plan an adaptation; and multiple execution components coordinate to execute an adaptation, e.g., they have to synchronize their adaptation actions. Decentralized control contrasts with central control. In central control, a single component exists (for one of the activities of self-adaptation) that performs its function. For example, analysis and planning is centralized in a self-adaptive system if this system has one analysis and one planning component that decides about when and how to perform an adaptation.

From this perspective, the functions of adaptation in a self-adaptive system (monitoring, analysis, planning, execution) can in principle be centralized or decentralized, independently of how the software of the managed and managing subsystems are deployed. However, in practice, when the managed software is deployed on a single node, the managing software will often also be deployed on that node and the adaptation functions are typically centralized. Similarly, fully decentralized adaptation functions typically go hand in hand with distribution of the software of the managed and managing subsystems. Between these two extremes, a variety of different ways to organize the functions of adaptation exist. The next two sections of this chapter elaborate on this.

3 A Notation for MAPE Patterns

As mentioned in the introduction, the adaptation logic (managed subsystem) typically involves feedback control loops with four key activities: Monitor (collect), Analyze (determine), Plan (prepare), and Execute (act), defining the classic MAPE control loop [28]. Given the central role control loops play in the way we conceptualize, design, and implement self-adaptive systems, [9] argues that "the design [of self-adaptive systems] must make the interactions of control loops explicit and expose how these interactions are handled".

Several authors, e.g. [46,38,9], have argued that existing approaches to describe software models are not well suited to represent control loops in the design. In [23], the authors introduce a UML profile for modeling control loops that extends UML modeling concepts. This approach allows control loops to become first-class elements of the design. The proposed UML profile supports modeling and reasoning about interactions between coarse-grained "controllers," while in this work we aim to model finer-grained interactions between the components of control loops.

In this section, we introduce a graphical notation to explicitly capture interacting MAPE loops by considering the control loop components M, A, P, and E, and their interactions. We call a recurring structure of interacting MAPE components a *MAPE pattern*.

In order to describe different MAPE patterns properly and overcome complexity, we introduce a unified, simple graphical representation based on a condensed notation of a MAPE loop as depicted in Figure 2. The key for this figure and the other figures with interacting MAPE loops in this chapter is described in Figure 3.

We distinguish between a MAPE pattern and an instance of the pattern. The former describes the abstract structure of the MAPE pattern in terms of abstract groups of MAPE components, the type of interactions between MAPE components between groups, and the interactions with the managed subsystem. The latter describes the concrete structure of the pattern for one particular configuration.

A group of MAPE components expresses a logical collection of MAPE components that may occur once or more in the pattern. The annotated cardinalities of the interactions between the groups of MAPE components determine the allowed occurrences of the different groups in the pattern. For the example pattern shown in Figure 2, there is only one occurrence of the group with four MAPE components allowed, while there are many occurrences possible for the group with only the M and E component (in the example shown at the bottom of Figure 2, there are two

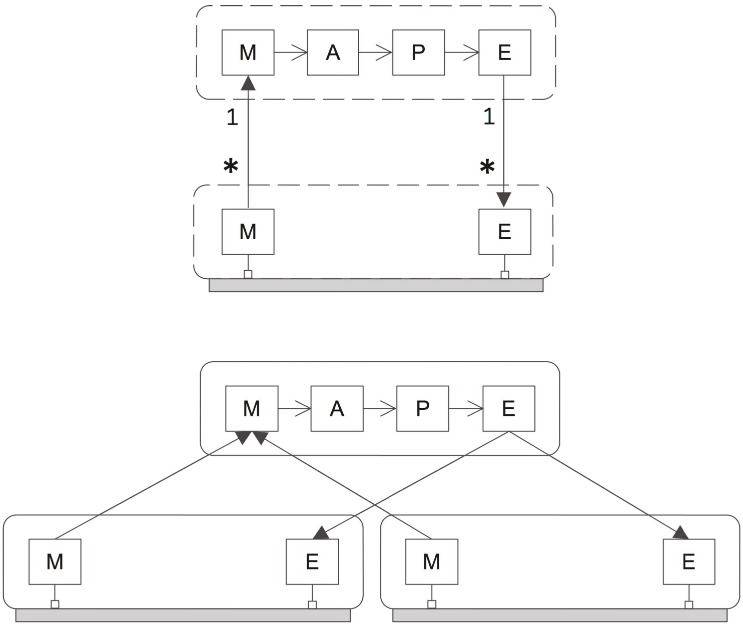


Fig. 2. Top: An example of a MAPE pattern. Bottom: an instance of the pattern in one concrete configuration.

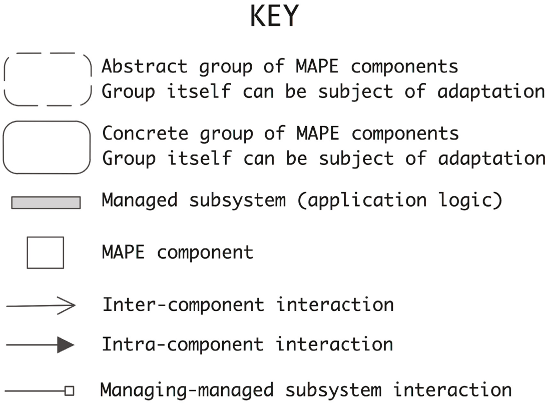

KEY
Abstract group of MAPE components
Group itself can be subject of adaptation
Concrete group of MAPE components
Group itself can be subject of adaptation
Managed subsystem (application logic)
MAPE component
Inter-component interaction
Intra-component interaction
Managing-managed subsystem interaction

Fig. 3. Key for patterns and instances

such occurrences). Notationally, groups partition a pattern so that it is easier to see which parts of the MAPE loop are being decentralized.

We differentiate the following types of interactions:

- Managed-managing subsystem interactions: these are the interactions between M components and the managed subsystem for monitoring purposes, and between E components and the managed subsystem for performing adaptations. Managed subsystem is the application logic that provides the system's domain functionality, or it can be a group of MAPE components that itself is subject of adaptation. Note that not every M and E component has to interact with the managed system. For example, in the instance shown in Figure 2, the M component in the top group is responsible for providing the required information about the managed subsystem to allow the A component to decide about adaptations. However, the actual collection of this information is delegated to M components that may reside at the different nodes where the managed subsystem is deployed.

– Inter-component interactions: these are the interactions between different types of MAPE components. In a typical MAPE loop, M interacts with A, A with P, and P with E. However, other interactions paths may be possible, such as subloops within a MAPE loop as discussed in [53].
– Intra-component interactions: these are the interactions between MAPE components of the same type, e.g., interactions between M components. Two important subtypes of this kind of interactions are delegation (as in the example pattern of Figure 2) and coordination. Coordination is used when components of the same type, but from different MAPE loops, interact with one another. Examples are two A components that have to coordinate to decide whether an adaptation should be applied, and two E components that have to synchronize the actions of an ongoing adaptation.

There are a number of important aspects of interacting MAPE loops that we do not consider explicitly in the notation and the patterns described in this chapter. First, we abstract away the knowledge aspect of MAPE components and how this knowledge is used and shared by the MAPE components. It is clear that knowledge exchange is an important design aspect of interacting MAPE loops and may have an impact on the applicability of the pattern. Second, we do not consider the distribution of the MAPE components and the communication resulting from actual deployment based on a particular network topology and supporting communication infrastructure (message oriented, publish-subscribe, etc.). Different deployments of the MAPE components may be possible, combined with different types of communication infrastructure, each with its particular benefits and trade-offs. We refrained from including these concerns in the patterns for the following reasons: (1) the treatment of knowledge heavily depends on the characteristics of the domain (e.g., the degree of cooperation or competition in the system, the sensitivity of particular knowledge, etc.), (2) the deployment of MAPE loops depends on constraints imposed by the underlying infrastructure (e.g., type of network, use of a particular middleware, etc.), and (3) including knowledge and deployment as first-class in the patterns would significantly expand the design space for each pattern and increase the complexity, and make less clear the interactions between the MAPE components, within and across MAPE loops. We touch upon a number of aspects of knowledge storage and exchange in section 6.

4 Patterns for Decentralized Control

We now present a selection of MAPE patterns that model different types of interacting MAPE loops with different degrees of decentralization. These patterns are not intended to be a complete enumeration of all possible configurations. In fact, the presented patterns emerged from the experiences of the authors with building self-adaptive systems, and discussions at a Dagstuhl seminar [44]. We use a standard template to present the patterns, consisting of the following parts: problem/motivation, solution, consequences, and examples.

We start by presenting two patterns, *coordinated control* and *information sharing*, both based on a fully decentralized approach that represents the antithesis with respect to a single centralized control loop. Both these patterns are based on a "flat" distribution model, where a multiplicity of peer MAPE loops operates in parallel to manage the overall system self-adaptation. Then, we present three other patterns, *master/slave*, *regional planning*, and *hierarchical control*, that are based instead on a "hierarchical" distribution model, where higher level MAPE components control subordinate MAPE components. The hierarchy generally reflects a separation of concerns among different control loops. These three patterns can be considered as intermediate points between fully decentralized and centralized control, as the root of the hierarchy basically constitutes a centralization point.

4.1 Coordinated Control Pattern

Problem/Motivation. In many cases, centralizing control for self-adaptation is simply not feasible. Among the possible reasons for this include: a) an inherent distribution of information in the system makes it too costly or even infeasible to collect all the data required for adaptation; b) due to the scale of the system the cost to process all the information at one place may be too high; and c) the system spans multiple ownership domains with no trustworthy authority to control adaptations. However, support for adaptation to achieve certain quality attributes is still desired. For example, multiple data centers still require guarantees that service-level agreements and legal regulations can be met, or control systems for managing traffic in a metropolitan area must coordinate to grant passage to emergency vehicles. In such cases, there may be no obvious way to organize control so that one part of the system has authority over another. In such a case, each control loop must coordinate with its peers to reach some joint decision about how to adapt.

Solution. A possible solution to overcome these problems is to decentralize the four MAPE activities. A MAPE loop is associated with each part of the managed system that is under its direct control. What characterizes this pattern is that basically all the M, A, P and E components of each loop coordinate their operation with corresponding peers of other loops. For example, A components exchange information to make a decision about the need for an adaptation, E components exchange messages to synchronize adaptation actions, etc. The

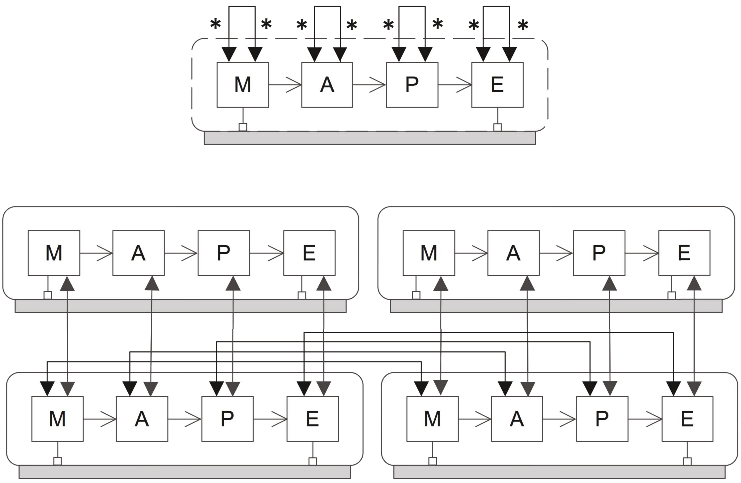


Fig. 4. Top: coordinated control pattern. Bottom: a possible instance.

interactions are typically localized, so that each component directly interacts with only a subset of its peers.

Figure 4 shows the decentralized control pattern and illustrates it for a concrete configuration. The pattern consists of one abstract group of MAPE components that contains all four components. The abstract group can be instantiated an arbitrary number of times. The pattern uses the standard sequence of interactions between the components within a MAPE loop, but variations may be possible. MAPE components of different MAPE loops can interact with peers to share particular information or coordinate their actions. The cardinalities of the intra-component interactions define the connectivity among MAPE components of the same type, that is, each component can interact with an arbitrary number of components of its own type.

The instance diagram at the bottom of Figure 4 shows a concrete instantiation of the pattern with four groups of MAPE components. Even if the pattern allows for a full connection among all peers, in a typical scenario, interactions among the same type of MAPE components will be localized.

Consequences. Decentralized control has the potential of good scalability with respect to communication and computation, depending on the coupling degree among peer components, and the number of other peers each MAPE component has to explicitly interact with. For systems in which adaptations can be performed based on local interactions between MAPE components, the communication overhead is limited to interactions with local peers. Furthermore, the computational burden is spread over the nodes. Decentralization may also contribute to improving robustness as there is no single point of failure. Decentralization of control may be the only option in cases where no single entity has the knowledge or authority to coordinate adaptations across a set of managed subsystems. There are a number of potential downsides of decentralized control as well. When coordination is required between MAPE components of many nodes, scalability may be compromised. The cost for reaching consensus about suitable adaptation actions may be high (in terms of communication and/or timing). Decentralized control may cause problems with ensuring consistency of adaptations. Furthermore, it may lead to sub-optimal adaptation decisions and actions, from the overall system viewpoint.

Examples. An example application that can be characterized as an implementation of the coordinated control pattern is presented in [18]. A component manager located at each node of a distributed application implements a logical control loop. The set of component managers cooperate to preserve some architectural constraints under certain events. All component managers rely on a group membership service and reliable broadcast to achieve a consistent view of the knowledge accumulated by their local M and A activities. Moreover, adaptation actions planned and executed by local P and E activities are globally coordinated by means of a totally ordered broadcast that implements a distributed locking scheme. As pointed out by the authors, the adopted mechanism to achieve global coordination requires explicit interaction among all MAPE loops. The resulting overhead thus limits the scalability of the proposed control architecture.

4.2 Information Sharing Pattern

Problem/Motivation. A software system consisting of a (potentially large) set of loosely connected components requires support for adaptation to maintain a particular concern or quality attribute. The components of the system are deployed on different nodes. Each part of the system can adapt locally, but requires information about the state of other nodes in the system because a local adaptation may impact these other nodes (e.g., on some quality attribute of those operations). However, apart from information sharing, nodes do not need to coordinate other adaptation activities. For example, in a sensor network for environmental monitoring (e.g., habitat monitoring), certain nodes may be equipped with sensors to detect a fire. In case a node detects a fire, it produces an alarm signal that can be spread effectively through the network using a smart gossip algorithm. Upon receiving the signal, nodes can activate a local adaptation procedure to anticipate disaster.

Solution. In contrast to the coordinated control pattern, the information sharing pattern restricts the inter-component type interactions of decentralized control to monitor (M) components only, as depicted in Figure 5. In particular, in this pattern only M components communicate with one another, while the A, P, and E components of each loop operate independently of their peers. The interactions are typically localized, that is, M components exchange information only with nodes in their (physical or logical) context. Thus, some information collected about the status of the managed systems is shared among the MAPE loops that allows local analysis, planning, and execution of adaptations

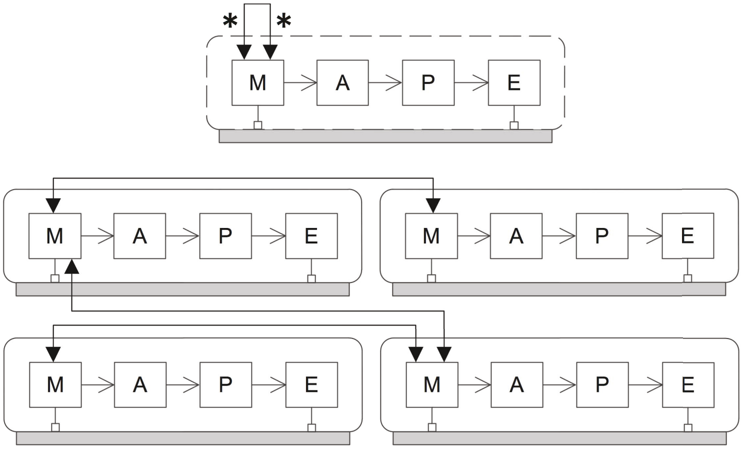


Fig. 5. Top: information sharing pattern. Bottom: concrete instance of the pattern.

without further coordination. Information sharing about the system state may be realized by explicit interactions among peer M components, or by implicit interactions where each M component independently monitors state information that is affected by the behavior of other nodes.

Figure 5 shows the information sharing pattern and illustrates it for a concrete configuration. The pattern consists of one abstract group of MAPE components containing all four components that can be instantiated an arbitrary number of times. At the inter-loop level, only M components can interact with an arbitrary number of peers to share particular information.

The instance diagram at the bottom of Figure 5 shows a concrete instantiation of the pattern with four groups of MAPE components. In this particular example, two M components interact with two peers (top left and bottom right), while the two other M components only interact with one peer.

Consequences. From a scalability perspective, information sharing may produce potentially higher benefits than coordinated control. Indeed, the less stringent interaction requirements (limited to M components only) may result in solutions that scale even better with respect to communication. However, this requires that the traffic between M components, in particular in case of explicit interactions, is limited in scope and volume. Another potential benefit is that since P, A, and E components can act locally without the need for coordination, this may lead to more timely decisions and execution of adaptations. On the other hand, the reduced coordination may increase locally optimal objectives, but at the cost of globally optimal ones. In the worst case, local decisions may conflict with one another, resulting in perpetual adaptation of the system, thus wasting resources and having adverse effect on the system's availability and stability.

Information sharing can be considered a special case of the coordinated control pattern discussed above, where the interactions among peer A, P and E components have been completely dropped. However, while the coordinated control pattern aims at directly achieving some regional or global objective through explicit cooperation among all types of MAPE components, the information sharing pattern adopts a different perspective, where achieving the global objective is less direct, because decisions are made locally rather than in a coordinated fashion. For this reason we prefer to give a "first-class citizen" status to this pattern in our list of alternative patterns.

Examples. The self-healing traffic monitoring system presented in [56] is an example in which the information sharing pattern is used to support self-healing in a traffic monitoring system by means of explicit state information sharing. The overall system consists of a set of cameras distributed along roads that are used to detect and report traffic jams (for example to a traffic light control system). A local traffic monitoring system deployed on each camera (i.e., the managed subsystem) monitors the traffic conditions in its viewing range. When a traffic jam is detected the local traffic monitoring systems form a dynamic organization with neighboring local traffic monitoring systems that span the range of the traffic jam. One of the monitoring systems is responsible for reporting the traffic jam to interested clients. To make the system resilient to camera failures, a self-healing subsystem is added to each local traffic monitoring system. To this end, the self-healing subsystems exchange information with self-healing subsystems on local cameras about their status using a ping-echo protocol. When a failure is detected (one of the self-healing subsystems does not respond with an echo message), the self-healing subsystem locally performs some analysis and planning activities that trigger local adaptation actions. Examples are removing the reference to a failed camera from the set of neighbors, and changing the dynamic organization of a set of monitoring cameras.

Another example of information sharing is described in [48] that aims to tackle the scalability problem of the group membership service and reliable broadcast used to preserve architectural constraints among distributed nodes as described in [18]. In this work, a gossip protocol is used to exchange information between nodes to support local adaptations. The authors show that the approach achieves a fault-tolerant and scalable solution to exchange information regarding the component configuration.

The two patterns identified and described so far are characterized by the introduction of different degrees of decentralization, but are both driven by a *flat separation of concerns* model, which places the different MAPE loops at the same conceptual/abstraction level. In other words, they play analogous roles on the different parts of the overall managed system they are directly responsible for. In the following we describe three hierarchical control patterns, where MAPE loops at different levels play different roles, with different responsibility levels.

4.3 Master/Slave Pattern

Problem/Motivation. There is a need to adapt a distributed software system for some concern. Monitoring and adaptations of the software needs to be done locally at each node, for example because of the high cost of transferring monitored data or because of the specificity of local adaptations. On the other hand, there is a need to provide global guarantees, predictability, and consistency about the state of the distributed system and its adaptations. For example, a central controller in an automated logistic systems (with cranes, conveyor belts, etc.) may rely on locally collected knowledge of machine software or their environment (which may include complex processing performed by M components) to trigger some of the machines to change their work mode (which may involve complex manipulations of the machine software performed by E components).

Solution. This pattern organizes the adaptation logic by creating a hierarchical relationship between one (centralized) master component that is responsible for the analysis and planning of adaptations (A and P activities) and multiple slave components that are responsible for monitoring and execution (M and E activities), see Figure 6. The pattern consists of two abstract groups of MAPE components. There is a single instance of the group with a P and an A component, and there can be an arbitrary number of instances of the group with an M and an E component. Each M interacts with the A component and P interacts with each E component. As such the pattern supports the typical flow of inter-component interactions of a MAPE loop, but with multiple instances of M and E components.

The M components of the slaves monitor the status of the local managed subsystems and possibly their execution environment and send the relevant information to the A component of the master. A, in turn, examines the collected information and coordinates with the P component, when a problem arises that requires an adaptation of local managed systems. The P component then puts together a plan to resolve the problem and coordinates with the E components on the slaves to execute the actions to the local managed subsystems.

The instance diagram at the bottom of Figure 6 shows a concrete instantiation of the pattern with three slaves.

Consequences. The master/slave pattern is a suitable solution for application scenarios in which slave control components need to process monitored information to derive the required data allowing centralized decision making, and execute local adaptation (probably based on higher-level adaptation instructions). On the positive side, centralizing the A and P components facilitates the implementation of efficient algorithms for analysis and planning aimed at achieving global objectives and guarantees. However, sending the collected information to the master component and distributing the adaptation actions may impose a significant communication overhead. Moreover, the solution may be problematic in case of large-scale distributed systems where the master may become a bottleneck. Finally, the master component continues to represent a single point of failure.

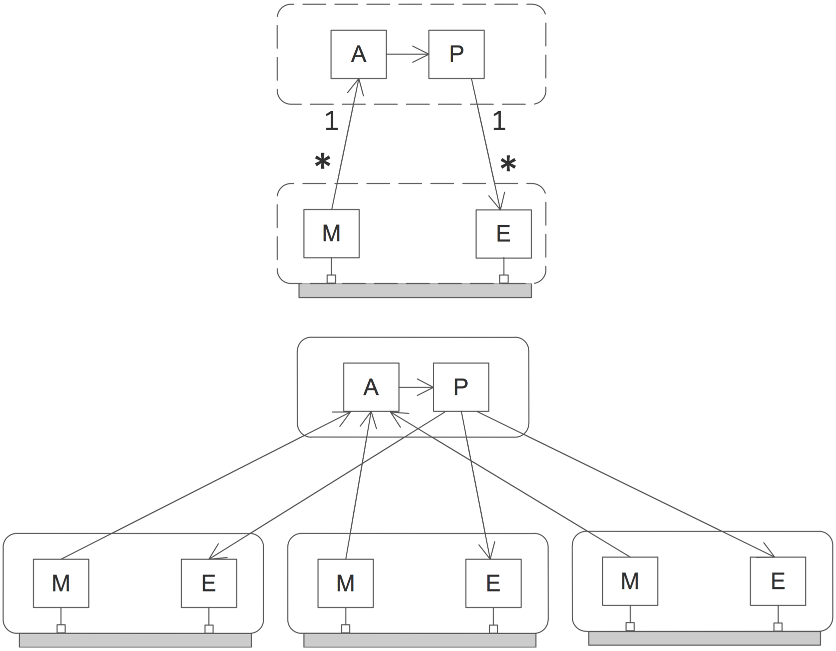


Fig. 6. Top: master/slave pattern. Bottom: concrete instance of the pattern.

Examples. The control architecture proposed in the RESERVOIR project [50,16] is an example of the master/slave pattern. The architecture is proposed in the context of a virtualized data center, where virtual execution environments are offered on top of a set of distributed physical servers. To meet the SLAs negotiated with the data center users, the control system monitors the system status (e.g., utilization degree of physical resources) through a set of monitors located at the different servers. A central master controller collects and analyzes these data, and plans suitable adaptation actions (that include, for example, changing the balance of the time slice among different virtual environments hosted by the same server, or live-migrating processes among physically separated servers).

The master/slave pattern is also at the basis of the control architecture for the *Znn.com* example system [10], developed according to the Rainbow framework [17]. The Znn.com system implements a news service that provides multimedia news content to its users, and is architected as a dynamically variable number of servers that serve clients requests by accessing a backend database. According to the Rainbow framework, the control system of Znn.com uses a distributed set of *probes* and *gauges* to monitor the system status. Collected data is centrally analyzed by an *architecture evaluator* that detects possible problems, while an *adaptation manager* decides on the best adaptation whose goal, in this example, is to keep the response time within a given threshold. The execution of adaptation actions (that include changing the number of active servers, and varying the "fidelity" of the provide responses) is then delegated to a set of distributed *effectors* driven by a *strategy executor*.

4.4 Regional Planning Pattern

Problem/Motivation. Different loosely coupled parts of software (regions) of a complex integrated software system want to realize local adaptations (within a region) as well as adaptations that cross the boundaries of the different parts (between regions). A typical scenario is a federated cloud infrastructure where adaptations within regions may aim to optimize resource allocation, while the objective of adaptations between regions may be delegation of certain loads under particular conditions (that owners of regions may not want to expose). Another setting is a supply chain management system where partners in the chain have certain local adaptation objectives, while adaptations between partners or system-wide adaptations may aim to achieve some global utility objective.

Solution. Regional planning provides one P component (a regional planner) for each region. A regional planner collects the necessary information from the underlying subsystems under its supervision to plan adaptations. Regional planners interact with one another to coordinate adaptions that span multiple regions.

Figure 7 shows the regional planner pattern and illustrates it for a concrete configuration. The pattern consists of two abstract groups of MAPE components, which both can occur an arbitrary number of times. The first group contains M, A, and E components. The second group contains only a P component. Inter-component type interactions follow the logical flow of a MAPE loop. Intra-component type interactions are restricted to P components.

The instance diagram at the bottom of Figure 7 shows a concrete instantiation of the pattern with two regional planners. For each region, the M components monitor the status of local managed subsystems and possibly the execution environment, the local A components analyze the collected information, and report the analysis results to the associated regional planner. The regional planner may then decide to perform a local adaptation (i.e., within the region), or regional planners may interact with one another to plan adaptations that span the two regions. Once the planners agree on a plan they can put the adaptations to action by activating the E components of the respective component groups involved in the adaptation.

Consequences. Regional planner enables a *layered separation of concerns* among different MAPE loops within a single ownership domain, where several MAPE loops delegate the planning function to a higher level component. For systems that cross the boundaries of ownership domains, regional planner enables a further (flat) separation of concerns for the planning function, where each planner is responsible for the planning of adaptations in its region. Local analysis of monitored data may reduce the amount of data and frequency of interactions with the planner. A downside of regional planner may be a lack of efficient adaptations. Aggregating the results of local analysis and coordinating the planning of adaptations may incur considerable overhead. Moreover, the pattern may require very detailed planning of the execution of adaptations as it does not support runtime coordination between E components.

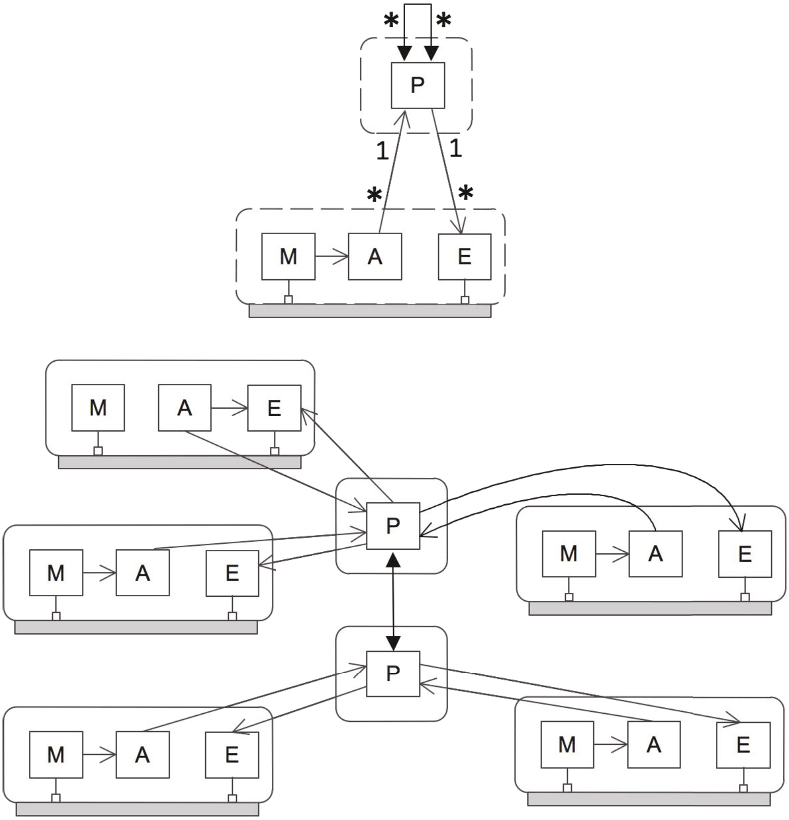


Fig. 7. Top: regional planner pattern. Bottom: concrete instance of the pattern with two regions.

Examples. The MOSES framework [8] is an example that instantiates the regional planner pattern. Within the context of service-oriented systems, the goal of MOSES is to provide a brokering service that supports runtime adaptation of composite services offered to multiple users with different service levels. The MOSES framework consists of a set of distributed monitoring components (*WS Monitor* and *QoS Monitor* components) that collect data about the availability and quality of service of different pools of candidate services that can be used to build the composite service managed by MOSES. Collected data are locally analyzed. The result of the analysis can trigger the calculation of a new plan by a centralized planning component (*Optimization engine*) that calculates a new abstract-to-concrete services binding policy, that is then realized at the endpoints.

The Deployment Improvement Framework [35] provides the ability to determine the optimal deployment of a software system at runtime and effecting it through runtime redeployment and adaptation of its components. This framework has been realized using a regional planner pattern, supporting redeployment in mobile and pervasive computing environments. In this particular case, each host has a *decentralized planner* (i.e., a regional planner) that only manages a single instance of a group of M, A, E components. Furthermore, each host has a *local monitor*, *local analyzer*, and *local effector* that are responsible for the monitoring, assessing changes in the monitored parameters, and redeployment of the components on the host they reside. Each host has a *model* that contains some subset of the system's overall model, populated by the data received from the local monitor and the model of the hosts to which this host is connected. The local analyzer on each host determines when the conditions for an improved deployment architecture occur, based on the local model. The decentralized planner then synchronizes with its remote counterparts to find a common solution. If the planners agree, the improved deployment architecture is effected by the local effectors.

4.5 Hierarchical Control Pattern

Problem/Motivation. The control architecture for a complex distributed system may itself become a complex system that needs to be adapted. In this case it is often necessary to consider multiple control loops within the same application. The loops can work at different time scales and manage different kind of resources, and resources with different localities. However, in this context, control loops need to interact and coordinate actions to avoid conflicts and provide certain guarantees about adaptations. The problem is then how to separate concerns to manage this complexity? Examples of such systems are: a) within a single data center, higher level control loops are responsible for achieving power consumption or workload goals, whereas local control loops manage workflow distribution between localized subsets of the nodes, and b) adaptation in pervasive computing environments could be organized into controllers that manage adaptation of human tasks as a user's goals change (in the order of minutes) and controllers that manage particular instances of these tasks to provide fault tolerance (in the order of seconds).

Solution. The hierarchical control pattern provides a *layered separation of concerns* to manage the complexity of self-adaptation. This pattern structures the adaptation logic as a hierarchy of MAPE loops. Different layers typically focus on different concerns at different levels of abstraction, and may operate at different time scales. Loops at lower layers operate at a short time scale, guaranteeing timely adaptation concerning the part of the system under their direct control. Higher levels operate at a longer time scale with a more global/strategic vision. MAPE loops at the bottom layer are directly concerned with different parts of the managed subsystem. MAPE loops at intermediate layers are concerned with the adaptation layers beneath. Finally, the MAPE loop at the top is concerned with the overall adaptation objectives of the system.

Figure 8 shows the hierarchical control pattern and illustrates it for a concrete configuration. The pattern is shown for a hierarchy of three layers, but more intermediate layers are possible. The M and E components of abstract groups at the bottom layer directly interact with the managed subsystem. M and E

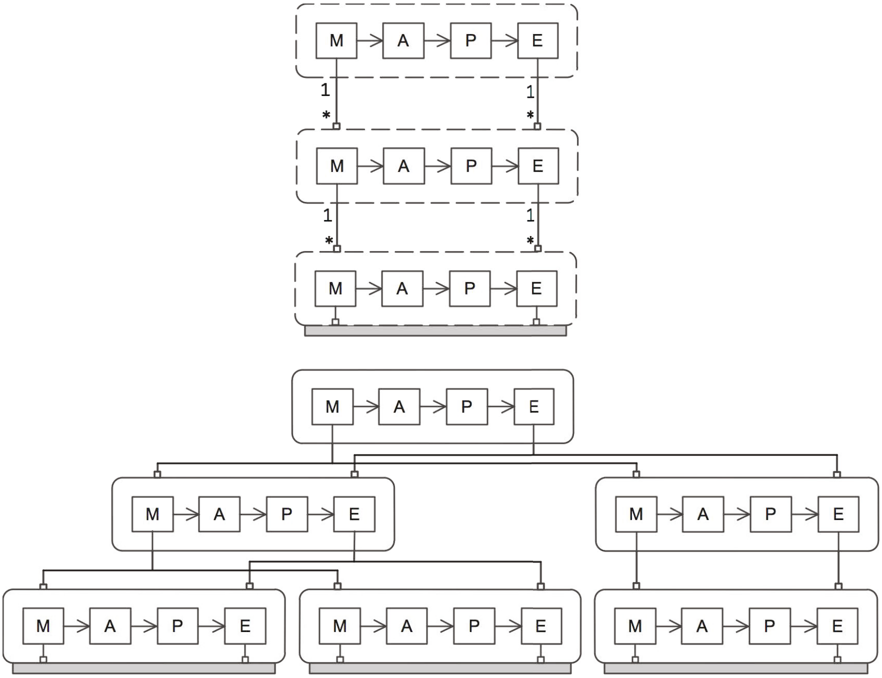


Fig. 8. Top: hierarchical control pattern. Bottom: concrete instance of the pattern.

components of abstract groups of higher-level layers interact with groups at the layers beneath.

The instance diagram at the bottom shows a concrete instantiation of the hierarchical control pattern. In this particular example, the hierarchy consists of three layers with two intermediate MAPE loops, one of them managing two subordinate loops, the other one managing a single loop.

Consequences. The hierarchical control pattern enables adaptation logic to be structured so that the complexity of self-adaptation can be managed. The hierarchical structure allows bottom layer control loops to focus on concrete adaptation objectives while higher level control loops can take increasingly broader perspectives. This corresponds to the layered organization of self-adaptation as proposed in [31]. However, there are a number of potential trade-offs with hierarchical control. The hierarchical decomposition of the adaptation concerns and the allocation of these concerns to different control loops might be difficult to achieve, in particular when goals interfere with one another. Moreover, it is known from behavior-based architectures [2] that the design and management of hierarchies with multiple layers can become very complex. As a result, there might be no guarantee that the overall solution meets the specifications.

Examples. A classic example of hierarchical control of adaption is the IBM architectural blueprint [25]. This approach consists of autonomic managers that add self-* properties to resources and these managers are, in turn, managed by other autonomic managers. At the highest level a manager takes high-level policies from users and delegates these throughout the hierarchy of autonomic managers. [4] discusses how the Autonomic Computing Reference Architecture (ACRA) can be used to orchestrate a set of autonomic managers that share knowledge sources to realize adaptations of managed resources.

The use of multiple control loops is proposed also in [33], where Litoiu et al. propose a hierarchical framework to deal with autonomic systems where it is possible to consider different time scales and different kind of managed resources.

Another example of the application of the hierarchical pattern can be found in [27]. In this work, the authors present Mistral, a multi-level hierarchical self-adaptive system. Specifically, Mistral is presented for a large data center environment and deployed in the form of a hierarchical control scheme with multiple instances of Mistral controllers managing different subsets of hosts and applications and operating at different time-scales. The controllers at the lower level manage a small number of machines and the applications hosted on them, while at the next higher level, a controller manages machines owned by multiple lower level controllers. Mistral reconfigures the system when variations in the monitored workload are detected and the adaptation actions are selected according to a predefined utility function.

5 Drivers for Selecting Control Schemas for Adaptation

So far we have outlined a set of patterns for decentralizing self-adaptive control loops, discussed forces that express conflicts among concerns when applying the patterns, and described how the patterns have been used by existing self-adaptive systems. Based on these insights, we discuss some of the drivers that should be considered by designers of self adaptive systems when choosing a MAPE pattern. As with any design, it is not possible to fulfill all requirements of all stakeholders with any one pattern. This means that choosing a pattern will depend on the relative importance of the requirements that stakeholders place on the managed system.

In the literature, it is usually quality concerns such as fault handling, efficiency, resource consumption, and load balancing that are the main goal of, and thus the main drivers for, self-adaptive solutions [24,52,55]. Due to the variability of domains and requirements, we cannot give an exhaustive list of how requirements may influence the choice of control mechanisms. Rather, we discuss in a few examples how certain kinds of requirements may impact this choice.

Optimization of one or more system properties is easier in centralized approaches where all measurement data is collected in one place, and only one entity makes control decisions based on that data. In decentralized approaches where several entities make local control decisions likely it will be more difficult to find a global optimum for system reconfiguration, since it is possible for control decisions to adversely influence each other, leading to frequent antagonistic adaptations. Ensuring other global properties is also easier to achieve with a centralized controller, when all data relevant to decision making is directly accessible. However, ensuring that this data is consistent in a distributed system poses significant challenges in itself [15,19].

The *scalability* of systems with respect to communication can be impacted significantly by the choice of a centralized or decentralized solution for self-adaptation. The larger and more complex an adaptive system becomes, the more data has to be processed to make control decisions. This data may also have to be transmitted from the node in the network where it is gathered to the node that hosts the decision logic. Scalability is thus impacted by at least two factors: the amount of data that has to be processed to make control decisions, and the amount of data that has to be transmitted across networks. In both cases, the more data there is, the less scalable the system will be. Decentralized systems can improve scalability if decisions can be made locally, based on data collected from the local context, or possibly subsets of the global monitoring data. Thus decentralization of self-adaptation functions may reduce the amount of data that has to be transmitted, and the amount of data that has to be processed to make decisions about adaptations. Effectively, this parallelizes adaptation decisions at the cost of making it hard to ensure global optima.

Robustness against node and link failure is the classical domain of distributed, replicated systems. A system with centralized control has a central control node as a bottleneck and potential single point of failure. Decentralized systems on the other hand will still be able to function even when some nodes and links fail. Only the nodes affected directly by link failures or controlled by a crashed controller will be affected in this scenario.

Responsiveness to changes needs to be considered. Different MAPE patterns have different reaction characteristics and MAPE loops in particular patterns may work at different time scales. For example, Rainbow (master/slave) can act on a system within seconds, but in some cases reactions in less than a second may be needed. To soundly organize MAPE loops hierarchically means that a loop must act at a time scale greater than its subordinate loops. Decentralizing control may make an adaptive system more responsive, but at the cost of producing subobtimal adaptations.

Different *administrative domains* may force particular types of adaptation control on a designer. For example, building a centralized model of the entire system may be infeasible if the knowledge of parts of the system has to be kept hidden (e.g., for strategic reasons). Similarly, an adaptive system may not be able to exercise control over some parts of the system. Consider a globally distributed data center network, where each data center may control itself, but cannot request reconfiguration in sibling centers because they are owned by different companies, or are under regulations of different governments. In such a case, the particular patterns that can be used for decentralization of control will be affected by the amount of information that is shared between the domains, and the amount of control that one domain can influence on another.

Domain constraints may also impose restrictions on the choice of a MAPE pattern for controlling adaptation. For example, in certain domains (e.g. banking), security or confidentiality requirements might prevent the sharing of data needed for control decisions with a central entity. In such a case, it might be feasible for subsystems to summarize and filter data so that no confidential data leaks, and then pass that data on to a centralized controller. Alternatively, a regional planning solution where each part of the overall system only deals with its own confidential data is conceivable. In some domains, for example mobile network applications, network interruptions and topology changes are so frequent that centralized solutions would be infeasible. In both the above scenarios it is infeasible or at least impractical to collect all relevant data at a central node and thus in these scenarios a decentralized solutions are more likely to be effective.

6 Discussion

The focus of the patterns described in this chapter is on the structures of MAPE loop components and their interactions. We have abstracted away the representation of knowledge in the patterns, how this knowledge is used and shared among the MAPE components, and how the system components are actually deployed on hardware. However, the ways in which knowledge is stored in the system and exchanged among MAPE components and the actual deployment of the system are important design concerns that will affect the applicability of the patterns. As explained in Section 3, we have refrained from including these concerns in the patterns since the way knowledge is treated and components are deployed heavily depend on the characteristics of the domain. Considering these concerns explicitly would increase the complexity of the descriptions of the patterns significantly. Instead, we consider the way knowledge is stored and exchanged between MAPE components and the distribution of the various components as two different *views* in the design of a self-adaptive system, complementary to the structured, interaction-oriented view of the patterns presented in this chapter.

In this section, we touch upon some aspects of knowledge in the design of MAPE loops of self-adaptive systems. It is our aim to give some initial ideas about such design decisions and their implications. Clearly, extensive research is required to treat the aspects of knowledge and deployment in a systematic manner. Concretely, we will look at two alternative approaches to deal with knowledge in the hierarchical control pattern.

As we explained in Section 4, one particular objective of the hierarchical control pattern is to manage complexity of self-adaptation by separating concerns of the adaptation logic in the form of a hierarchy of MAPE loops. Figure 8 shows the interactions among MAPE loops in consecutive layers. Here, we show two possible approaches to share knowledge among MAPE components in this pattern.

Figure 9 shows an instance of the hierarchical control pattern with individual knowledge repositories for each MAPE loop. In this configuration, knowledge can only be exchanged via the interactions of MAPE loops of consecutive layers. Figure 10 shows an alternative configuration with additional knowledge repositories that are shared among MAPE loops within layers.

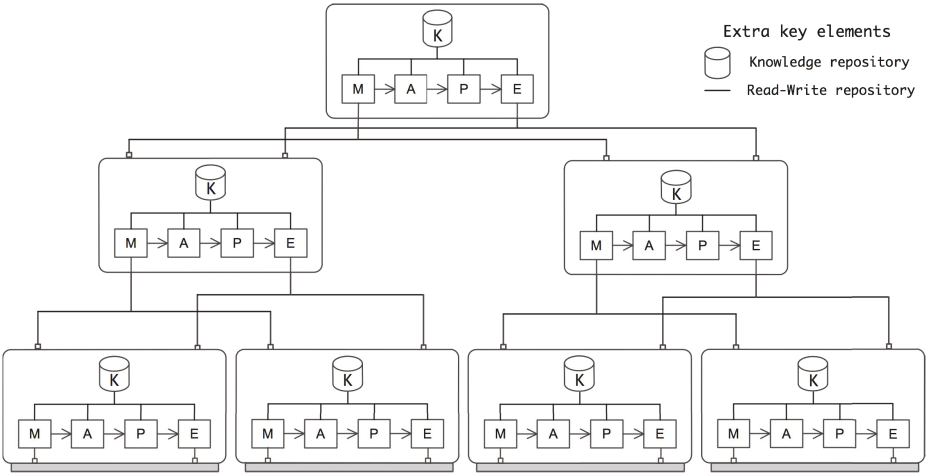


Fig. 9. Instance of the hierarchical control pattern with knowledge repositories per MAPE loop

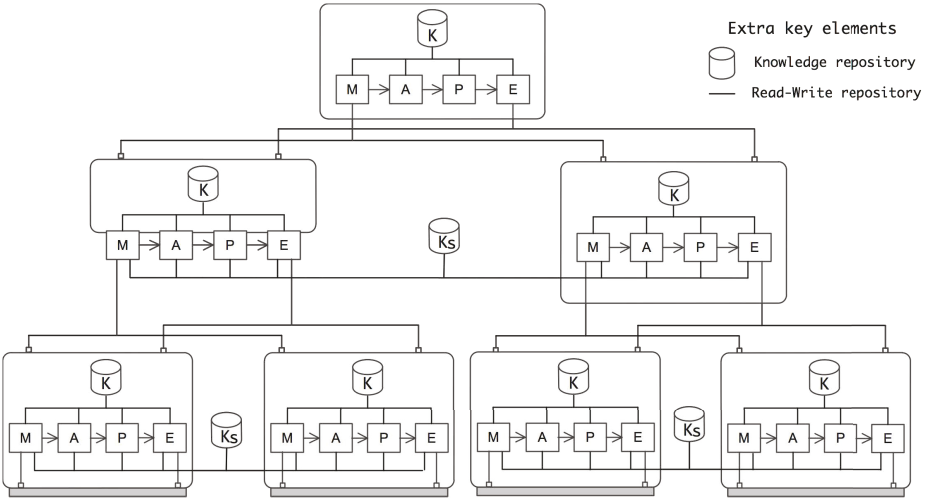


Fig. 10. Instance of the hierarchical control pattern with additional shared knowledge repositories within layers

In the first approach, each MAPE loop maintains knowledge in a local repository. This approach restricts the exchange of knowledge between MAPE loops of consecutive layers. Such knowledge exchange is important, for example, to enable higher level MAPE loops to make decisions about adaptations at lower levels. In the second approach, MAPE loops can also exchange knowledge with siblings using a shared knowledge repository. This approach enables MAPE loops at one layer to coordinate adaptations without direct interference of MAPE loops at the layer above. Shared knowledge in the form of a shared tuple space, for example, creates a loose coupling between MAPE loops at one layer. Such an organization may be a solution to situations where adaptations have to be realized between managed subsystems that are connected in a very dynamic manner, e.g., in a mobile setting.

These two example scenarios illustrate that the aspect of knowledge can be treated in (potentially many) different ways for this particular pattern, resulting in specific variants of the pattern that are useful for different domains with different characteristics and specific requirements. Study of these variants for different types of MAPE patterns is an interesting area of future research.

7 Related Work

The work on software architecture and design patterns is extensive. The series on Pattern-Oriented Software Architecture (POSA) by Buschmann et al. [7,45,29,6], covers fundamental patterns [7], like *Reflection* and patterns specific for a domain, e.g., resource management [29], concurrency [45], and distribution [6]. These patterns provide concrete strategies and mechanisms to address specific architectural or implementation problems. The patterns proposed and described in this chapter are different in that they are considering the structure and interaction of MAPE loops and their components at an abstract level. On a more concrete level, various POSA patterns are premier candidates to realize such patterns.

Research in self-organizing systems have brought forward a number of patterns for distributed decentralized computing, for instance to support replication, which are inspired from biology [3]. Compared to the architecture-centric perspective presented in this chapter, the biology inspired patterns are described from an algorithmic perspective with more precise behavioral semantics. Such patterns are more related to the aspect of knowledge (see section 6), and are candidates to realize particular types knowledge exchange in some of the MAPE patterns.

There is a large body of work in designing and implementing self-adaptive systems, and subsequent recent reflection by researchers to develop advice and patterns for them. Gomaa and Hussein [21] have developed several software reconfiguration patterns for dynamic evolution of software architectures. They define a *reconfiguration pattern* to be a set of recurring sequences of adaptation steps (e.g., stopping/starting, (un)linking, adding/removing) necessary for ensuring *consistent* adaptation of a software system. To ensure the dynamic replacement of a component does not jeopardize the systems consistency, a reconfiguration pattern first places the component in the *quiescent* state [30], before replacing it at runtime. Subsequent to this work, several approaches have shown the utility of reconfiguration patterns to achieve consistency during adaptation. In [22], Gomaa et al. employ reconfiguration patterns in the context of self-managed service-oriented software systems, while in [14], Esfahani et al. show their utility in the design of architecture-based middleware solutions. The patterns described in this chapter are different, as we have aimed to distill patterns that result from the different compositions of MAPE components in the managing system, while reconfiguration patterns deal with ensuring the consistency of the managed system during adaptation.

Ramirez and Cheng [42] describe a set of design patterns for building dynamic software systems. Their patterns are at the level of software design, and aim to facilitate the construction of a self-adaptive software system. The purpose of patterns proposed in their work is to help engineers to better understand alternative means of achieving runtime adaptation in the system's design. The patterns proposed in this chapter are at a higher level of granularity, as we adopt an architecture-centric perspective with the aim of better understanding the impact of decentralization on self-adaptive software systems.

Some of the co-authors have defined a formal reference model (FORMS) that can be used to understand and reason about self-adaptive systems, formally defining the relationships among the environment, managing system, and managed system [57]. This model provides three perspectives of self-adaptive systems, among one is a distribution perspective that offers an abstract representation of interacting MAPE loops in terms of coordination mechanisms. Work in this chapter concentrates on the relationships between MAPE components (which refines the discussion in FORMS), and does not really consider in detail the relationship between these elements and the managed system and environment.

Explicit representation of control loops in self-adaptive systems has been discussed in [23]. A UML profile for modeling control loops is presented which allows the modeling of sensors, actuators, controller, and their interactions as parts of the adaptation logic. They are able to model a variety of instances of self-organizing systems with mutliple control loops. The work in this chapter translates the abstract concept of controller in [23] into concrete patterns of interacting MAPE loops, and provides a platform for discussing the trade-offs of applying different patterns.

8 Conclusions and Challenges Ahead

In this chapter, we have laid the groundwork for consolidating knowledge on decentralized control in self-adaptive systems in the form of patterns of interacting MAPE loops. We derived these patterns from their use in practice, introduced a notation for describing them, and discussed their ramifications with respect to certain quality attributes. This work can be used as a basis for understanding different patterns of decentralized control by software engineers of self-adaptive systems, and for comparing work in the field.

As this chapter represents only the start of the work on decentralization of control in self-adaptive systems, we conclude this chapter with a number of research challenges ahead, to contribute to the research road-map in the field. We start with more concrete challenges and move towards long term visions at the end.

Include state/knowledge. Currently, the patterns cover only structural aspects of decentralization of control in self-adaptive systems. As an important future challenge, data/knowledge aspects should also be covered in the patterns, including the differentiation between global and local knowledge. In particular, different forms of partitioning and/or (full/partial/lazy) replication of knowledge should be seamlessly included in the MAPE patterns, as they provide another path of indirect interaction between the MAPE elements. For example, one could let the components in a hierarchical MAPE loop interact by shared knowledge as described in section 6 or by introducing a new hybrid control pattern by using the coordinated control pattern in the middle layer of the hierarchical control pattern for achieving a similar interaction. One particularly interesting approach to including knowledge in the patterns for decentralized control is by defining a complementary view of the managed system that focuses on the knowledge concern.

Adding behavior and communication. The patterns presented in this chapter focus on centralization vs. decentralization of the primary functions of self-adaptation with MAPE loops. Future research should focus on identifying and classifying (i) the behavior of each MAPE component (for example, filtering or preprocessing monitored information to minimize data exchange; using decentralized or self-organized planning algorithms), (ii) the communication paradigms used for the various interactions in the patterns (for example, direct message exchange; use of a blackboard for coordination), and (iii) the specific protocols used for communication between the MAPE components (for example, push-pull, request-reply, negotiation). The pattern notation introduced in this chapter could be improved by adding different connector types between the elements to take care of items (ii) and (iii).

MAPE activities beyond sequence. In this chapter, we assume the activities in the MAPE loop follow in sequence (i.e., Monitoring followed by Analysis, Planning, and finally Execution). It is conceivable that there may be interactions that do not follow this logical sequence. For example, analysis and planning may coordinate, or analysis might coordinate with monitoring to insert new monitors or request information more or less frequently. [53] is an example in which coordination between MAPE components organized in sub-loops within a MAPE-loop is studied. Nested loops [20] are another approach where the managed system of the outer loop comprises the managing inner loop plus the system managed by the inner loop. A systematic study of MAPE activities beyond a traditional sequence is an interesting area that should be studied further. A related challenge is to study how the style of the managed system might have implications on the architecture of the managing system.

Extending the architectural expressiveness of our patterns. The notation used in this paper could be extended with a formal foundation. This would enhance the expressiveness of the patterns and allow precise expression and reasoning about different configurations of the patterns. Additionally, a formal model would enable analysis of certain properties of systems modeled with the patterns. Such analysis is particularly important for decentralized self-adaptive systems in which global properties are often a critical aspect of the design. One effort in this direction is FORMS [57] that provides formally defined modeling elements (in the Z language) to specify architectures of managing subsystems, allowing to reason about the architectural characteristics of distributed self-adaptive software systems. However, this approach does not support fine-grained specifications of interacting MAPE loops.

Dealing with uncertainty. To perform proper adaptations, the managing subsystem needs runtime models, including models of (the relevant parts) of the managed subsystem and the environment in which the self-adaptive system is deployed. Such models may introduce uncertainty, for example caused by non-determinism in the environment, inconsistencies between the managed subsystem and its runtime representation, etc. Tackling the problems related to uncertainty is challenging [13], as the causes of uncertainty are often not under control of the designer. The situation is exacerbated in decentralized self-adaptive systems, where there is no central authority, and adaptation decisions have to taken based on partial knowledge. Dealing with uncertainty in self-adaptive systems that have multiple control loops is a challenging area for future research.

Standardization. So far, the research community has focused on standardizing the notification interfaces of sensors and effectors of managed subsystems, but ignored communication *within* the MAPE loop. For instance, the Oasis standard [39] defines events that are broadly understood by vendors of system management tools. Our position of making the decentralization of control loops explicit underlines the need for standardizing the interactions between the MAPE loop components. That includes interactions among MAPE components within a control loop as well as interactions between MAPE loop components of the same type of different control loops. This will comprise interface definitions (signatures and APIs), message formats, and protocols. The necessity of this standardization has already been appreciated in the past, e.g., in [32] the authors standardize the communication from the A to the P component by using a standard data exchange format (e.g., SOAP), but no comprehensive approach exists so far.

Control Theory. There are substantial theoretical foundations for understanding control systems in other engineering domains, embodied in control theory. In this chapter, we have defined patterns for how to assemble a particular kind of control loop (i.e., MAPE), but we have not discussed how theories and techniques from control theory apply to the control of self-adaptative systems. For example, it would be desirable to describe the transfer function of a managed system. A transfer function defines the relation between a controlled system's input and output, in particular how effectors affect subsequent sensor readings. Having such a function for the control loops of self-adaptive systems would mean that we could reason about the properties (such as stability) of the control loop being designed. Even more so, if the transfer function is available during run-time in a machine-processable way, this reasoning can be subject to run-time adaptation as well. The forms that a transfer function takes in different software domains has received scant attention. In the context of service-oriented computing, the transfer function could relate to service level agreements (SLA): sensor readings would be mapped to SLA values, so that the transfer function describes how the generic control loop needs to be controlled at the effector in order to result in the desired SLA behavior. This will probably include the mapping of certain SLAs to particular MAPE patterns that are proven to be effective with respect to these SLAs. Investigating how research from the models@runtime community can inform the definition of transfer functions for software would be a good starting point.

Adaptive coupling with mutable control patterns. Complexity theory [37] shows that the overall properties of a complex software system are largely determined by the internal structure and interaction of its parts and less by the function of its individual constituents [54]. Even more so, the internal structure of a system is formed by relationships of differing strengths between constituents. Components with tighter connections (or coupling) cluster to sub-systems, while other components may remain more loosely-coupled. Hence, a complex software system provides a mixture of tightly and loosely coupled parts. As an important consequence, the overall system properties (e.g., scalability) are determined not only by the structure but even more so by the strength of coupling of its relationships [20]. Our control patterns support different forms of coupling. For example, the information sharing pattern provides a much looser form of coupling compared to the decentralized control pattern, thus the former potentiality scales much better than the latter.

In order to use the full potential of the extended architectural expressiveness, e.g., with nested control loops, the outer loop should be able to control the strength of coupling of the inner loop. This means nothing else than "switching" from one pattern to the other during operation. Future research should investigate approaches like [41] to allow for mutable control patterns.

Pattern enumeration and application. The patterns described in the chapter do not fully enumerate all possible decentralization patterns, and in fact the patterns could potentially be combined in any number of ways (for example, in federated data centers the information sharing pattern could be used to manage adaptation between data centers, while a hierarchical pattern could be used within a data center). Future work should look at a broader range of self-adaptive systems to enumerate all the patterns that have been used successfully in practice.

Furthermore, understanding when it is best to use one pattern over another should be an active area of future research. We conceive of at least three dimensions that will affect the choice of pattern:

1. The desired quality attributes and the level of guarantee required for them. For example, it may be easier to prove that global quality attributes such as performance will be achieved in the master/slave pattern, but that scalability would be difficult to achieve.
2. The architecture of the managed system will likely influence which patterns are applicable. For example, a hierarchical pattern will be unlikely to work

if there is no obvious hierarchy of authority in the managed system, or applying the information sharing pattern will likely be influenced by how much information about the managed subsystems can be shared.

3. Domain constraints may affect the choice of a particular pattern. For example, centralizing adaptation decisions may not be possible for confidentiality reasons or because of dynamics in the network topology. In such scenarios, a decentralized solution may be preferable.

We expect that a better understanding of how the drivers relate to the patterns, and how the architecture of the managed system restricts the patterns that can be employed to manage it, will lead to more principled design of self-adaptive systems in the future.

Acknowledgments. This chapter is based on the results of a collaborative effort of a breakout group for and the Dagstuhl Seminar on Software Engineering or Self Adapative Systems, October, 2010. The authors would like to thank the other participants in the breakout session for their contributions to the discussion: Schahram Dustdar, Jeff Kramer and Rick Schlichting, as well as other attendees of the seminar.

References

1. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Modeling Dimensions of Self-Adaptive Software Systems. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 27–47. Springer, Heidelberg (2009)
2. Arkin, R.: Bahavior-Based Robotics (1998)
3. Babaoglu, O., Canright, G., Deutsch, A., Caro, G.A.D., Ducatelle, F., Gambardella, L.M., Ganguly, N., Jelasity, M., Montemanni, R., Montresor, A., Urnes, T.: Design patterns from biology for distributed computing. ACM Trans. Auton. Adapt. Syst. 1, 26–66 (2006)
4. Brittenham, P., Cutlip, R.R., Draper, C., Miller, B.A., Choudhary, S., Perazolo, M.: It service management architecture and autonomic computing. IBM Syst. J. 46, 565–581 (2007), `http://dx.doi.org/10.1147/sj.463.0565`
5. 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 2007), Minneapolis, MN, USA (May 2007)
6. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing, vol. 4. Wiley, Chichester (2007)
7. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture, A System of Patterns, vol. 1. Wiley, Chichester (1996)
8. Cardellini, V., Casalicchio, E., Grassi, V., Lo Presti, F.: Adaptive Management of Composite Services under Percentile-Based Service Level Agreements. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 381–395. Springer, Heidelberg (2010)

9. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: 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)
10. Cheng, S.W., Garlan, D., Schmerl, B.R.: Evaluating the effectiveness of the rainbow self-adaptive system. In: SEAMS, pp. 132–141 (2009)
11. Dobson, S., Denazis, S., Fernndez, A., Gati, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A survey of autonomic communications. ACM Transactions Autonomous Adaptive Systems (TAAS) 1(2), 223–259 (2006)
12. Dowling, J., Cahill, V.: The K-Component Architecture Meta-model for Self-Adaptive Software. In: Matsuoka, S. (ed.) Reflection 2001. LNCS, vol. 2192, pp. 81–88. Springer, Heidelberg (2001)
13. Esfahani, N., Kouroshfar, E., Malek, S.: Taming uncertainty in self-adaptive software. In: SIGSOFT FSE, pp. 234–244 (2011)
14. Esfahani, N., Malek, S.: On the Role of Architectural Styles in Improving the Adaptation Support of Middleware Platforms. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 433–440. Springer, Heidelberg (2010)
15. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985)
16. Gambi, A., Pezzè, M., Young, M.: SLA protection models for virtualized data centers. In: Proc. of the Int. Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS (2009)
17. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer 37, 46–54 (2004)
18. Georgiadis, I., Magee, J., Kramer, J.: Self-Organising Software Architectures for Distributed Systems. In: 1st Workshop on Self-Healing Systems. ACM, New York (2002)
19. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33, 51–59 (2002), http://doi.acm.org/10.1145/564585.564601
20. Goeschka, K.M., Froihofer, L., Dustdar, S.: What soa can do for software dependability. In: Workshop on Architecting Dependable Systems (WADS 2008), Supplemental Proceedings of the 38th IEEE International Conference on Dependable Systems and Networks (DSN 2008), pp. D4–D9. IEEE Computer Society (2008)
21. Gomaa, H., Hussein, M.: Software reconfiguration patterns for dynamic evolution of software architectures. In: Proceedings of Fourth Working IEEE/IFIP Conference on Software Architecture, WICSA 2004, pp. 79–88 (2004)
22. Gomaa, H., Hashimoto, K., Kim, M., Malek, S., Menascé, D.A.: Software adaptation patterns for service-oriented architectures. In: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC 2010, pp. 462–469. ACM, New York (2010)
23. 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, SOAR 2010, pp. 21–28. ACM, New York (2010)
24. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing–degrees, models, and applications. ACM Computing Surveys 40, 7:1–7:28 (2008), http://doi.acm.org/10.1145/1380584.1380585

25. IBM: An architectural blueprint for autonomic computing. Tech. rep., IBM (January 2006)
26. Jackson, M.: The meaning of requirements. Ann. Softw. Eng. 3, 5–21 (1997), `http://dl.acm.org/citation.cfm?id=590564.590577`
27. Jung, G., Hiltunen, M.A., Joshi, K.R., Schlichting, R.D., Pu, C.: Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In: Proceedings of the 2010, IEEE 30th International Conference on Distributed Computing Systems, ICDCS 2010, pp. 62–73 (2010)
28. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
29. Kircher, M., Jain, P.: Pattern-Oriented Software Architecture. Patterns for Resource Management, vol. 3. Wiley, Chichester (2004)
30. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Trans. Softw. Eng. 16, 1293–1306 (1990), `http://dl.acm.org/citation.cfm?id=93658.93672`
31. 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)
32. Leymann, F.: Combining Web Services and the Grid: Towards Adaptive Enterprise Applications. In: Castro, J., Teniente, E. (eds.) First International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA 2005) - CAiSE Workshop, pp. 9–21. FEUP Edi cões (June 2005)
33. Litoiu, M., Woodside, M., Zheng, T.: Hierarchical model-based autonomic control of software systems. In: Proceedings of the 2005 Workshop on Design and Evolution of Autonomic Application Software, DEAS 2005, pp. 1–7. ACM (2005)
34. Maes, P.: Computional reflection. Ph.D. thesis, Vrije Universiteit (1987)
35. Malek, S., Beckman, N., Mikic-Rakic, M., Medvidovíc, N.: A Framework for Ensuring and Improving Dependability in Highly Distributed Systems. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems III. LNCS, vol. 3549, pp. 173–193. Springer, Heidelberg (2005)
36. Malek, S., Mikic-Rakic, M., Medvidovic, N.: A decentralized redeployment algorithm for improving the availability of distributed systems. In: 3rd International Conference on Component Deployment, Grenoble, France (November 2005)
37. Manson, S.M.: Simplifying complexity: a review of complexity theory. Geoforum 32(3), 405–414 (2001)
38. 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, ULSSIS 2008, pp. 23–26. ACM, New York (2008), `http://doi.acm.org/10.1145/1370700.1370707`
39. OASIS, `http://www.oasis-open.org`
40. 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 Intelligent Systems 14, 54–62 (1999), `http://dx.doi.org/10.1109/5254.769885`
41. Pereira, J., Oliveira, R.: The mutable consensus protocol. In: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, pp. 218–227. IEEE Computer Society (October 2004)
42. 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, SEAMS 2010, pp. 49–58. ACM, New York (2010)

43. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2), 1–42 (2009)
44. Schloss Dagstuhl Seminar 10431, Wadern, Germany: Software Engineering for Self-Adaptive Systems (October 2010), http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=10431
45. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture. Patterns for Concurrent and Networked Objects, vol. 2. Wiley, Chichester (2000)
46. Shaw, M.: Beyond objects. ACM SIGSOFT Software Engineering Notes (SEN) 20(1), 27–38 (1995)
47. Shaw, M., Clements, P.: The golden age of software architecture. IEEE Softw. 23, 31–39 (2006), http://dl.acm.org/citation.cfm?id=1128592.1128707
48. Sykes, D., Magee, J., Kramer, J.: Flashmob: distributed adaptive self-assembly. In: Proceeding of the 6th International Symposium on Software Engineering for Adaptive and Self-managing Systems, SEAMS 2011, pp. 100–109. ACM, New York (2011), http://doi.acm.org/10.1145/1988008.1988023
49. Tanenbaum, A.S., van Steen, M.: Distributed Systems: Principles and Paradigms, 2nd edn. Prentice-Hall, Inc., Upper Saddle River (2006)
50. Toffetti, G., Gambi, A., Pezzè, M., Pautasso, C.: Engineering Autonomic Controllers for Virtualized Web Applications. In: Benatallah, B., Casati, F., Kappel, G., Rossi, G. (eds.) ICWE 2010. LNCS, vol. 6189, pp. 66–80. Springer, Heidelberg (2010)
51. Van Dyke Parunak, H., Brueckner, S.A., Sauter, J.A., Matthews, R.: Global convergence of local agent behaviors. In: Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS 2005, pp. 305–312. ACM, New York (2005)
52. Villegas, N.M., Müller, H.A., 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, New York (2011), http://doi.acm.org/10.1145/1988008.1988020
53. Vromant, P., Weyns, D., Malek, S., Andersson, J.: On interacting control loops in self-adaptive systems. In: Proceedings of Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011), Honolulu, Hawaii (2011)
54. Wegner, P.: Why interaction is more powerful than algorithms. Commun. ACM 40(5), 80–91 (1997)
55. Weyns, D., Iftakhir, M.U., Malek, S., Andersson, J.: Claims and supporting evidence for self-adaptive systems: A literature review. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2012. ACM, New York (2012)
56. Weyns, D., Malek, S., Andersson, J.: On decentralized self-adaptation: lessons from the trenches and challenges for the future. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2010, pp. 84–93. ACM, New York (2010), http://doi.acm.org/10.1145/1808984.1808994
57. Weyns, D., Malek, S., Andersson, J.: Forms: Unifying reference model for formal specification of distributed self-adaptive systems. ACM Transactions on Autonomous and Adaptive Systems, Special Issue on Formal Methods for Pervasive, Self-Aware, and Context-Aware Systems 7(1) (2012)

Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems

Gabriel Tamura[1], Norha M. Villegas[2], Hausi A. Müller[3], João Pedro Sousa[4],
Basil Becker[5], Gabor Karsai[6], Serge Mankovskii[7], Mauro Pezzè[8],
Wilhelm Schäfer[9], Ladan Tahvildari[10], and Kenny Wong[11]

[1] University of Lille 1-LIFL-INRIA, France,
Los Andes University and Icesi University, Colombia
gabriel.tamura@inria.fr

[2] University of Victoria, British Columbia,
Canada, and Icesi University, Colombia
nvillega@cs.uvic.ca

[3] University of Victoria, British Columbia, Canada
hausi@cs.uvic.ca

[4] George Mason University, USA
jpsousa@gmu.dot.edu

[5] Hasso Plattner Institute at the University of Potsdam, Germany
basil.becker@hpi.uni-potsdam.de

[6] Vanderbilt University, USA
gabor.karsai@vanderbilt.edu

[7] CA Inc., Canada
serge.mankovskii@ca.com

[8] University of Milano Bicocca, Italy and University of Lugano, Switzerland
mauro.pezze@unisi.ch

[9] University of Paderborn, Germany
wilhelm@upb.de

[10] University of Waterloo, Canada
ltahvild@uwaterloo.ca

[11] University of Alberta, Canada
kennyw@ualberta.ca

Abstract. Software validation and verification (V&V) ensures that software products satisfy user requirements and meet their expected quality attributes throughout their lifecycle. While high levels of adaptation and autonomy provide new ways for software systems to operate in highly dynamic environments, developing certifiable V&V methods for guaranteeing the achievement of self-adaptive software goals is one of the major challenges facing the entire research field. In this chapter we (i) analyze fundamental challenges and concerns for the development of V&V methods and techniques that provide certifiable trust in self-adaptive and self-managing systems; and (ii) present a proposal for including V&V operations explicitly in feedback loops for ensuring the achievement of software self-adaptation goals. Both of these contributions provide valuable starting points for V&V researchers to help advance this field.

R. de Lemos et al. (Eds.): Self-Adaptive Systems, LNCS 7475, pp. 108–132, 2013.

1 Introduction

Software validation and verification (V&V) concerns the quality assessment of software products throughout their lifecycle. Its goal is to ensure that the software product satisfies its functional requirements and expected quality attributes [1–3]. Over the past decade, many self-adaptive approaches and systems have been proposed by researchers from the software engineering for adaptive and self-managing systems (SEAMS) community, with multiple adaptation purposes [4, 5]. Certainly, many of the proposed self-adaptive software (SAS) systems have been designed to operate in highly dynamic socio-technical ecosystems where requirements, models, and contexts change at runtime [6]. This wide spectrum of system types, adaptation concerns, and dynamic goals has made it difficult to develop general runtime V&V methods. Unsurprisingly, V&V of SAS systems running in safety-critical environments is particularly challenging [7].

For inherently non-adaptive systems, that is, systems based on stable and well-known system execution conditions, many V&V methods, techniques and tools have been developed to be applied at design time. However, the quality assessment of SAS systems is challenging, not only because their adaptation objectives may vary according to environmental conditions at runtime, but also because the systems evolve to satisfy their evolving dynamic requirements. In this realm, V&V tasks—traditionally applied at design-time—are required to certify structural and behavioral aspects in the different phases of the adaptation process. In addition, these tasks must be performed at runtime in the two essential parts of a SAS system, namely, the adaptation mechanism, and the target system.

Besides the SEAMS community, there are several other communities dealing with runtime V&V, although not necessarily for SAS systems. During the past decade, the real-time verification (RV) community has run a workshop concerned with the monitoring and analysis of system executions.[1] The longer term goal of RV, already stated at RV 2001, is to investigate whether the use of lightweight formal methods applied during the execution of programs is a viable complement to the current heavyweight methods proving programs' correctness always before their execution, such as model checking and theorem proving, among others. Dynamic analysis, or the analysis of data gathered from a running program, has great potential for self-adaptive systems because it relies on direct monitoring mechanisms that expose the system's actual behavior [8]. The Models@runtime workshop, which emerged from the model-driven software development community, aims to use model-driven techniques for validating and monitoring runtime behavior. The Requirements@runtime workshop—collocated with the Requirements Engineering (RE) conference—aims to explore the potential of runtime abstractions and models of requirements, to be used as a practical means to address the challenges posed by volatile or poorly-understood environmental contexts.[2] In many ways, these workshops and conferences focus on different aspects

[1] International Conference on Runtime Verification
`http://runtime-verification.org/`

[2] `http://www.comp.lancs.ac.uk/~bencomo/RRT/`

of runtime V&V such as requirements, models, properties, instrumentation, and dynamic analysis.

Naturally, for the non-adaptive parts of a self-adaptive system, the traditional V&V methods can be used effectively. For the adaptive parts, runtime V&V methods are needed to guarantee self-adaptation objectives, independently of what is adapted. In general, SAS systems feature mechanisms based on the idea of the feedback loop [9]. We aptly termed the foundational science for runtime V&V methods *control science.* Control science can be defined as a systematic way to study certifiable V&V methods and tools to allow humans to trust decisions made by self-adaptive systems. In a 2010 report, Dahm identified control science as a top priority for the US Air Force (USAF) science and technology research agenda for the next 20 years [10]. Certifiable V&V methods and tools are critical for the success of autonomous, autonomic, smart, self-adaptive and self-managing systems.

One systematic approach to control science for adaptive systems is to study V&V methods for the mechanisms that sense the dynamic environmental conditions and the target system behavior, and act in response to these conditions by answering the questions *what*, *when* and *how* to adapt. In this paper we use the answers to these questions as key factors to determine *when* and *where* to perform V&V activities in the context of feedback loops—the common core of SAS systems.

This roadmap chapter focuses on research challenges concerning runtime V&V for the adaptive parts of self-adaptive systems in highly dynamic environments. In particular, we (i) analyze the cases in which the objectives, the system, or the monitoring infrastructure of a SAS system must be adapted; and (ii) propose how to make V&V tasks explicit in the feedback loop model elements, using results obtained by the aforementioned communities in this research field. Our goal is to provide researchers with a vision of open challenges in V&V for SAS systems, and discuss opportunities not only for proposing new runtime V&V techniques, but also for building on top of existing ones. In addition, our proposal for the explicitness of V&V tasks provides solid starting points for V&V researchers from other communities to deploy different techniques and methods for improving the trustworthiness of self-adaptive and self-managing systems.

The remaining sections of this chapter are organized as follows. Section 2 describes a concrete industrial case study that we use to illustrate the concepts, concerns and challenges discussed in this chapter. Section 3 outlines several challenges that arise from the differences in V&V requirements between software developed with traditional methods and self-adaptive software, and presents selected V&V drivers for self-adaptive software. Section 4 presents a refinement of the general feedback adaptation loop to propose a model that explicitly involves V&V tasks to address some of the previously identified challenges. This section also presents approaches from SEAMS-related research communities that provide valuable contributions for the assessment of SAS systems. Finally, Section 5 concludes the chapter.

2 Application Example

This section introduces our concrete industrial case study.[3] In this application example, self-adaptation is exploited to implement SOA governance mechanisms to enforce service level agreements (SLAs), such as those on performance, availability and confidentiality, in a cloud computing infrastructure [11]. In SOA and cloud-based systems QoS are highly affected by, and dependent on changing situations. On the one hand, SLAs may be violated at any time during system execution due to changes in the situation of relevant context entities such as computational infrastructure components and users. On the other hand, as businesses and users' requirements are evolving continuously, contracted QoS conditions (i.e., adaptation goals) may be frequently re-negotiated.

In this example, a performance SLA has been negotiated in terms of three throughput service level objectives (SLOs) to guarantee three different levels of system capacity in a cloud-based e-commerce platform: normal, medium and high load. These SLOs are observed on the bottleneck-operation of the system, *ProcessingPurchaseOrders*, and measured in terms of number of transactions per time unit. A *normal capacity* is required for a regular load of the shopping platform. A *medium capacity* is required when special offers are placed on social networks promoting them. A *high capacity* must be guaranteed to deal with the highest peak load of the platform caused by shopping seasons such as "Black Friday".

Governing the efficiency of the service-oriented infrastructure to optimize operational costs is a major concern for the retailers of this example. Hence, a self-adaptive mechanism based on service component architecture (SCA) reconfiguration was implemented to ensure quality of service (QoS) requirements in the service-oriented system. The *adaptation goal* for this dynamic service-oriented infrastructure is the contracted system capacity, in terms of the performance SLA. *Short settling time* and *consistency* correspond to the *adaptation properties* to be preserved. Adaptation properties in this application example are borrowed from the catalog proposed by researchers from the SEAMS community [5]. Settling time is defined as the time required for the adaptive system to achieve the desired state. Consistency guarantees the structural and behavioral integrity of the managed system with respect to the respective SCA integrity constraints [12], after its adaptation.

Use Case 1: Controlling the Elasticity of the E-Commerce Platform. As efficiency is a major concern, the capacity of the system must be either increased, or decreased according to the context situations that determine the expected load of the system. To accomplish this, context monitors must keep track of the popularity of special offers placed on social networks, as well as the day of the year to determine the applicable shopping season.

[3] This example is based on the IBM Centre for Advanced Studies (CAS) Canada project: "Managing Dynamic Context to Optimize Smart Interactions and Smart Services" `https://www-927.ibm.com/ibm/cas/cassis/viewReport?REPORT=747.`

Use Case 2: Re-negotiating Adaptation Goals at Runtime. After the e-commerce platform has been in operation, a new set of SLOs is added to the performance SLA. These new SLOs define different thresholds of response time that must be guaranteed according to the classification of the e-commerce platform's customers. Customers are classified as regular and premium users. A particular *maximum response time* threshold applies to regular customers. For premium customers, the maximum response time must correspond to 90% of the threshold defined for regular customers. Response time thresholds can be re-negotiated at runtime.

This example is used in the following sections to discuss runtime V&V concerns and research challenges in SAS systems, and illustrate the need for applying V&V tasks at runtime.

3 V&V Drivers for Self-Adaptive Software Systems

In this section we analyze and discuss drivers or key factors to consider when performing V&V tasks for SAS systems. We identify these drivers by (i) comparing how V&V for software that is adaptable at runtime differs from V&V for software that is immutable at runtime; and (ii) analyzing concerns that arise when dealing with three types of context changes that have been addressed by SAS systems, namely, in the objectives, the system, and the monitoring infrastructure to be adapted.

The goal of this section is twofold. First, it analyzes the classic V model for software development, and in particular its V&V activities, from the perspective of SAS systems. Second, it presents V&V drivers that we identified by analyzing three foundational questions concerning assurances for SAS systems: *when* to perform V&V tasks? *what* must V&V tasks validate and verify? and *where* in the adaptation cycle must these V&V tasks be performed? In light of these drivers, we identify research avenues in the form of research problems and opportunities to integrate V&V methods and techniques into the engineering of self-adaptive software systems.

3.1 The Classic V Model for System Development

Figure 1 illustrates how V&V activities are enacted in traditional software engineering to ensure that, at the different levels of system development, the software satisfies a given set of requirements.

This set of requirements is usually specified in advance of system development, allowing the definition of the corresponding complete *problem space*. From these requirements, a *solution space* is delimited and a solution derived and conceived in the form of a software architecture, which is refined into a software design. Both, architecture and design, are expressed as *models* (formal, semi-formal or informal), which can be verified at design time on their functional properties (e.g., correctness) with respect to the initial set of requirements. From this design, the software is materialized as units of code, which are gradually integrated, verified and tested

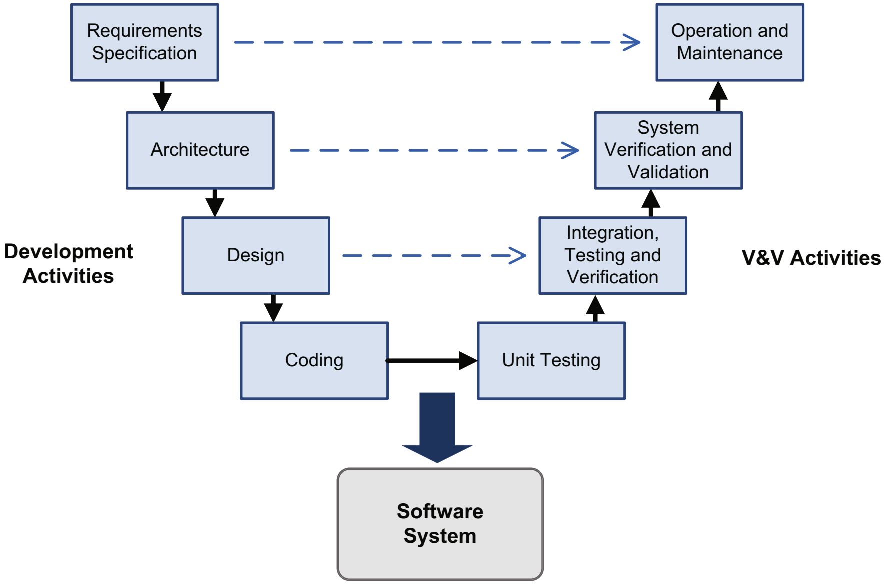

Requirements Specification
Operation and Maintenance
Architecture
System Verification and Validation
Development Activities
Design
Integration, Testing and Verification
V&V Activities
Coding
Unit Testing
Software System

Fig. 1. The classic V model for system development (adapted from [13]). Each development phase is subject to a corresponding V&V phase—horizontal layers—as the software is built and integrated.

until the final system is obtained. Finally, this system is verified and validated as a whole before its deployment in production environments [14].

In this general lifecycle, the software quality is guaranteed by different V&V strategies applied in its different phases, even though several variations may provide additional assurances. For instance, despite the described flow of activities following the solid arrows in the depicted V-model, the dashed arrows allow V&V to be performed on the artifacts produced by any of the development activities (e.g., requirements or design models).

Among the V&V strategies that have been used, the software testing methods are the ones most commonly used in industry. Software testing methods can be very effective both in revealing failures and assessing the reliability of software systems, but cannot provide evidence of the absence of faults [13, 15]. More rigorous and effective strategies to reason about the program correctness employ model checking, graph-based, and other model-based software testing and verification methods [16–19]. However, these V&V methods have focused generally on design time. Therefore, the assessment and certification of system properties after changes occurring during system execution, either for ensuring the satisfaction of changing requirements, or for re-certifying system properties after adaptation, require not only traditional V&V methods adjusted to be applied at runtime, but also the adoption of non-traditional ones to be applied in the different adaptation phases of SAS systems.

Another important difference between these two types of software systems is the lifecycle phases in which V&V tasks are performed. In the classic V model, V&V tasks are performed by software developers before deployment into production environments. However, in the adaptation process, the system architecture, design and implementation are evaluated, reconsidered and reconfigured at runtime by the system itself, according to relevant context changes. Hence, V&V tasks must be performed by the adaptation mechanism during the adaptation process. This has three important consequences. First, after the software initial release is in execution, the software development lifecycle phases (i.e., architecture, design and implementation) are in fact "absorbed", at least to some extent, by the adaptation (i.e., self-reconfiguration) process. Second, the boundaries between these phases, now amalgamated in the adaptation process, are blurred [20]. Third, the target system may be adapted and reach a state that was unforeseen at design time, and thus, the system has not been verified for that state. In simple ("switching") systems with a few possible adaptation results, this can be verified at design-time, but for a system with a very large number of resulting states this is unfeasible. Therefore, for SAS systems, in addition to the V&V tasks performed at design time, the system itself needs to apply different V&V methods at runtime. At this point, two questions arise: (i) what V&V methods are the most adequate to be applied at runtime? and (ii) at which specific moments in the adaptation process should these methods be applied?

These are certainly challenging questions, given that, additionally, many aspects of self-adaptive systems are impossible to assess at design time, due to their strong dependency on the actual execution environments. A recent US Air Force research agenda posits that developing certifiable V&V methods for highly adaptive systems is one of the major challenges facing the entire field. Understanding the inherent properties of adaptation mechanisms for software systems, and the ways in which these properties can be guaranteed may require a large part of the decade, if not more [10]. In the following sections we address these questions.

3.2 The Viability Zone of Self-Adaptive Software Systems

We define the *viability zone* of a SAS system as the set of possible system states in which the system operation is not compromised [21]. That is, the set of states where the SAS system's requirements and desired properties (i.e., adaptation goals) are satisfied. Viability zones can be characterized in terms of relevant context attributes and corresponding desired values. These context attributes correspond to either measurements of internal variables of the target system or the adaptation mechanism, or environmental variables whose variations can take the system outside its viability zone. A particular SAS system may have more than one associated viability zone (e.g., one for each adaptation goal). The global viability zone of a SAS system thus results from the composition of these partial viability zones. Moreover, existing viability zones can be added, replaced or adjusted by adding or removing variables of interest at runtime.

In the case of our application example, the initial viability zone is defined in terms of the performance SLA, and the three throughput SLOs (normal, medium and high capacity—cf. Use Case 1 in Sect. 2). These three SLOs constitute three different levels of system capacity that can been interpreted as three viability sub-zones. The variables that characterize the e-commerce platform's viability zone correspond to the actual throughput of the *ProcessingPurchaseOrders* operation, the popularity of special offers placed in a social network (including whether an offer has been placed), and the shopping season, all of them to be monitored at runtime. Seasons are characterized in three groups: regular, medium (e.g., Valentine's Day), and high seasons (e.g., Christmas and Black Friday). Another associated viability zone in this example is used to control the short settling-time adaptation property. This zone is defined by a single-variable that is monitored to keep track of the time the e-commerce platform takes to reconfigure the system to obtain the desired throughput. Furthermore, after the re-negotiation of the performance SLA, a new viability zone must be computed at runtime to control the response time SLOs (cf. Use Case 2).

V&V under Viability Zone Dynamics. It can be argued that our definition of viability zone coincides with that of the *solution space* used in traditional (i.e., non-adaptive) software systems. However, from the previous examples it is clear that the viability zone can change with context changes, as opposed to the solution space concept, which is assumed to be fixed.

In effect, the viability zone of a target system under adaptation constantly varies along adaptation dimensions. These variations take place every time the adaptation operation modifies either the target system architecture (e.g., adding or removing components and connectors) or the controller itself (e.g., modifying its parameters or replacing the control algorithm), thus introducing new, or removing existing variables and associated domain types.

Therefore, not only are runtime V&V methods required to cope with the viability zone dynamics problem, but these V&V methods also need to be automatically generated according to the modifications that result from dynamic adaptation. Thus, to extend the V&V coverage of the expanded viability zone, runtime models are required for the incremental derivation of software artifacts for V&V monitoring and checking.

In the aforementioned example, understanding its viability zone dynamics is crucial for the self-adaptive e-commerce platform V&V tasks. In fact, the adaptation mechanism together with its V&V tasks can be interpreted as an optimization problem, where the optimal solution is chosen among those within the viability zone, based on the system capacity policies, as proposed by Balasubramanian *et al.* [22]. First, transitions between viability sub-zones are associated to an adaptation policy (adaptation strategy). For instance, when the system is approaching the threshold between a lower and a higher load—going from the lower to the higher, the corresponding adaptation task must be triggered to increase the system processing capacity accordingly (e.g., by deploying new components for scalable processing). Similarly, the system capacity must be reduced when it goes from a higher load to a lower one. In both cases, as the software component structure is modified as a result of the adaptation, the SCA structural conformance property must be verified at runtime on the resulting system. Second, changes in viability zones (e.g., changes in variables' thresholds, and addition or replacement of variables in adaptation dimensions) may affect not only the adaptation strategy, but also the monitoring infrastructure, since these changes are caused by changes in adaptation goals. Finally, runtime V&V tasks aim to keep the adaptive system inside its viability zone, even when viability zones are subject to changes at runtime. The way how V&V tasks contribute to achieve this goal depends on the nature of the system and its requirements. For instance, for safety-critical applications, runtime V&V must check if the system will trespass the boundaries of its viability zone as a result of an adaptation, before instrumenting it in the running system. In those cases where self-adaptation is interpreted as an optimization problem, V&V tasks can be used both, before the adaptation, and after it. Before the adaptation, to restrict the alternatives to consider, to those within the viability zone. After the adaptation, to ensure that the solution is satisfying the new requirements under possibly changed context situations.

3.3 What: Requirements and Adaptation Properties

We identified the underlying V&V questions in the domain of SAS systems as *what*, *where*, and *when* to validate. This subsection focuses on the *what* to validate question. The answer to this question relates to the identification of adaptation goals (e.g., non-functional requirements of the target system) and adaptation properties (e.g., desired characteristics of the adaptation mechanism). Explicit adaptation goals and properties are crucial for the specification of suitable V&V models for SAS systems, and the identification of corresponding metrics. Moreover, having an explicit mapping between adaptation goals and properties, and relevant context is required to ensure the coherence between V&V tasks and the relevant context variables that characterize the system's viability zone. In our application example, we address this mapping by defining *context-driven SLAs* [11]. As proposed in [11], context-driven SLAs are machine readable specifications of SLAs, in the form of contextual resource description framework (RDF)[4] graphs, that not only state contracted conditions explicitly (e.g., the throughput and response time SLOs), but also the context variables, and context monitoring strategies required to keep track of the system behavior and its viability zone (e.g, sensors and monitoring conditions to measure throughput, response time, settling time, and the popularity of special offers, as well as identify shopping seasons).

Properties and Metrics. V&V concerns for self-adaptation certification can be classified according to the two constitutive parts of a SAS system. The first relates to the certification of the target system, while the second to the certification of the adaptation mechanism [5]. After the 2010 Dagstuhl Seminar on Software Engineering for Self-Adaptive Systems, researchers from the SEAMS community conducted an extensive analysis of self-adaptive approaches and developed a framework for evaluating self-adaptive systems, where desired properties of the target system (i.e., adaptation goals) and the adaptation mechanisms (i.e., adaptation properties) are identified explicitly and defined in terms of quality attributes [5].

[4] `http://www.w3.org/RDF/`

Several of the identified adaptation properties were borrowed from control theory [9, 23] and re-interpreted for self-adaptive software. Moreover, they classified adaptation properties according to *how* and *where* these properties are observed (cf. Table 1). Concerning how they are observed, some properties can be evaluated using static verification techniques, while others require dynamic verification and runtime monitoring (i.e., runtime V&V). With respect to where they are observed, properties can be evaluated on either the target system, or the adaptation mechanism. However, most properties can only be observed directly on the target system even when they are used to evaluate the adaptation mechanism.

Table 1. Classification of adaptation properties according to how and where they are observed [5]

Adaptation Property	Property Verification Mechanism	Where the Property is Observed
Stability	Dynamic	Target system
Accuracy	Dynamic	Target system
Settling Time	Dynamic	Both
Small Overshoot	Dynamic	Target system
Robustness	Dynamic	Adaptation Mechanism
Termination	Static	Adaptation Mechanism
Consistency	Both	Target system
Scalability	Dynamic	Both
Security	Dynamic	Both

Having no well defined and explicit metrics that can be used to assess properties, it is impossible to realize the vision of runtime V&V. Nevertheless, even though the importance of having such explicit metrics seems obvious, an important barrier for the assessment of dynamic software systems is the lack of accurate metrics to evaluate adaptive software [4]. Therefore, more research is required on the definition of applicable domain-specific metrics that effectively provide the means for evaluating relevant properties of dynamic software systems. Some examples of metrics and corresponding mappings to adaptation properties used in actual self-adaptive implementations and research initiatives, where non-functional requirements are a major concern, are summarized in the evaluation framework for self-adaptive software proposed by Villegas *et al.* [5].

An important challenge for V&V of SAS is to investigate innovative mechanisms that enable the application of techniques such as model checking, compositional verification, program synthesis, and dynamic analysis and monitoring to asses these properties at runtime. Another important research concern is the management of trade-offs that may arise from the need to ensure multiple properties—trade-offs among multiple viability zones.

Dependency on Runtime Monitoring. Besides using different representation models for target system behaviour, traditional V&V also uses controlled simulation environments. However, given the difficulties for building models to predict self-adaptive system behavior for every possible operational situation, and the impossibility of characterizing these situations in simulation environments, V&V of context-dependent properties requires information gathered at runtime. For instance, in mission-critical systems, only with actual runtime measurements it is possible to determine confidently whether the target system is within its viability zone [24]. Understanding and characterizing which properties of self-adaptive software are critically dependent on runtime information is crucial for realizing V&V in SAS effectively.

Uncertainty in Self-Adaptation. Context dependent requirements usually involve uncertainty. Uncertainty can be both a challenge and an opportunity. In safety-critical systems uncertainty is a tough challenge that exacerbates verification tasks significantly [19, 24]. In other scenarios such as e-commerce applications, uncertainty is an opportunity, since the system can provide better service to customers by leveraging the context information that arise from the interactions between the users and the system, as well as from users' situations [25].

The adaptive nature of the execution environment in SAS systems makes uncertainty one of the most difficult challenges to be addressed by V&V researchers. An interesting research opportunity is to tailor feedback loop-based mechanisms used to manage uncertainty in modern control theory to context-aware SAS systems [26]. Similarly, the rich literature on engineering adaptive mechanisms for flight control systems inspires many researchers. In particular, Schumann and Gupta proposed a V&V method to calculate safety regions for adaptive systems around the current state of operation based on a Bayesian statistical approach [27]. With this approach, they can provide a confidence measurement on the probable accuracy of the system's model under a particular situation.

We argue for the exploitation of viability zones as useful mechanisms to manage uncertainty in the assurance of SAS systems. From this perspective, the management of uncertainty problem focusses on determining explicit boundaries for the SAS system's viability zones and controlling the target system accordingly (cf. Sect. 3.2).

3.4 Where: Separation of Concerns

We distinguish two system levels in SAS systems: the target system to be dynamically adapted according to context changes, and the adaptation mechanism. For runtime V&V it is critical to understand the extent of the separation of these two levels. This separation of concerns allows us to characterize, investigate, and analyze V&V research problems for self-adaptive software effectively, by focusing specifically on the respective concerns of each level.

Although the discussion in this chapter is applicable to both feedback and feedforward control in computing systems [9], we focus on feedback control since runtime V&V depends on online measurements from the target system and the adaptation mechanism. That is, measured outputs are important for making adaptive system quality decisions at runtime. Moreover, as feedforward control takes also environmental disturbances—external context—into account, subsequently we use the terms feedback loop and control loop interchangeably. Following the feedback loop abstraction from the V&V perspective, the target system is an open loop for which the adaptation mechanism provides the elements to close the loop. In other words, the target software system itself is unaware of both context conditions and self-performance, with respect to the satisfaction of its own functional and non-functional (context-dependent) requirements. Thus, given that the objective of V&V is to guarantee the quality of a system, and this quality is expressed as the fulfillment of its requirements, in SAS systems V&V tasks must be incorporated as part of the adaptation loop. This implies that, in addition to the common context monitoring elements considered in feedback adaptation loops, additional components dedicated to verification and testing of the target system itself are required. At runtime, these components could, for instance, perform partial and incremental model checking on the next most probable states with respect to the current system state. Referred to our application example, the property to be verified could be the structural conformance of the reconfigured software application, with respect to the SCA structural constraints.

In addition, the separation of concerns between the target system and the adaptation mechanism implies different possible V&V interactions among these two system levels. Each of these interactions affects, in different ways, the ultimate goal of self-adaptation: the continued and effective operation of the target system services under varying context conditions. Of course, a general requirement is that the adaptation mechanism executes as unobtrusively and independently as possible from the target system. This observation has two implications. First, the target system functionalities must execute uninterruptedly for as long as possible while the adaptation mechanism performs the required adaptations on these functionalities. Moreover, the target system is expected to remain functional even if the V&V fails (i.e., if it indicates that the new system state is invalid). This implies that the adaptation mechanism must also run without interruptions. Second, unavailability of the adaptation mechanism should not cause unavailability of the target system. However, at some point, it is reasonable to expect that the adaptation mechanism, and even the target system itself, will require a shut down for maintenance or correcting system failures.

3.5 When: V&V in the Adaptation Process

Traditional V&V strategies involve checking and testing before system deployment under presumably well-defined conditions of system operation. This process of checking and testing is often automated using model checking, theorem proving, and testing tools. For context-dependent requirements, traditional V&V activities and certification techniques, designed to be applied before system deployment on fully specified requirements, are neither sufficient nor applicable. On the one hand, these formal V&V methods are often too expensive to be executed regularly at runtime when the system adapts due to their time and space complexity. On the other hand, context-dependent variables are unbound at design time, but bound at runtime. Thus, performing V&V on these variables at runtime is valuable to reduce the verification space significantly, even when the

SAS system viability zone varies with context changes. From this perspective, it is crucial to determine precisely when in the adaptation process these V&V operations are to be performed to guarantee the system properties and prevent unsafe operation. As previously mentioned, the lack of effective runtime V&V methods is considered one of the biggest obstacles and major challenges for the wide adoption of self-adaptive software applications in industry [10].

In addition, the considerations discussed in the previous section (i.e., the where) require the analysis of at least the following questions with respect to *when* to perform V&V tasks:

i. What properties can be exclusively verified at design time (executing neither the target system nor the adaptation mechanism)?
ii. What properties can be exclusively verified or tested at system configuration time?
iii. What properties can be exclusively verified or tested at runtime?
iv. What properties can be verified or tested either at design time, configuration time, or at runtime?

For instance, a machine-learning-based adaptive mechanism, such as the one proposed by Elkhodary *et al.* [28], could be checked for training coverage with respect to pre-defined adaptation cases at configuration time, before its deployment in production. However, the effectiveness of learned adaptations should be verified at runtime, based on information gathered from the actual adapted system behavior.

The answers to these questions are highly interdependent. For example, an approach aimed at verifying stability (what)—a behavioral adaptation property of the adaptation mechanism—may require the assessment of performance quality factors such as latency, throughput and capacity [5]. These factors assume runtime (when) monitoring on the target system (where). Stability is defined as the convergence of the subject system behavior toward a desired state. Moreover, many of the design concerns, such as availability, performance, survivability, fault tolerance and security, are highly interdependent and evolve at discrete points in time. It is critical to separate these concerns at design as well as at runtime.

In our application example, the performance SLA and its SLOs (throughput and response time—cf. Use Case 1 and Use Case 2 in Sect. 2), as well as the settling time and SCA structural conformance properties constitute the *what* to validate. Regarding the *where* question, throughput and response time must be observed on the target system, settling time must be observed on both the adaptation mechanism and the target system, whereas the SCA structural conformance, on the target system. Finally, concerning the *when* question, V&V tasks to ensure these requirements and properties must be performed at runtime.

In the following section we give some answers to the *when* and *where* questions by extending the feedback-loop elements with V&V responsibilities.

4 Making V&V Explicit in the Self-Adaptation Loop

So far, we have analyzed four key V&V drivers for SAS systems that pose major research challenges for SEAMS-related communities: (i) the viability zone and its dynamics; (ii) what to validate and verify, and its dependency on context information; (iii) where to validate—closely related to the separation of concerns between the target system and the adaptation mechanism; and (iv) when to perform V&V in SAS with respect to the adaptation loop.

To advance SAS goal assurance, we argue for the integration of runtime V&V tasks in the adaptation process. Accordingly, this section presents our proposal for making V&V tasks explicit in the elements of feedback adaptation loops, as for example in the MAPE-K loop [29]. Moreover, we discuss runtime V&V enablers (i.e., requirements at runtime, models at runtime, and dynamic context monitoring), which provide effective support to materialize V&V assurances for self-adaptation. Our proposal, depicted in Fig. 2, clearly answers *when* and *where* concrete V&V tasks can be implemented in the adaptation loop, using these enablers. The V&V enablers—dashed boxes in this figure—also provide a guide for other SEAMS-related research communities to contribute with runtime V&V methods for SAS systems. With this proposal we contribute to the convergence of these research communities towards the realization of suitable assurance mechanisms for SAS systems.

Applying this proposal to our application example, we use requirements at runtime to represent machine-readable specifications of the performance SLA, and its throughput and response-time SLOs. In this way, runtime validators and

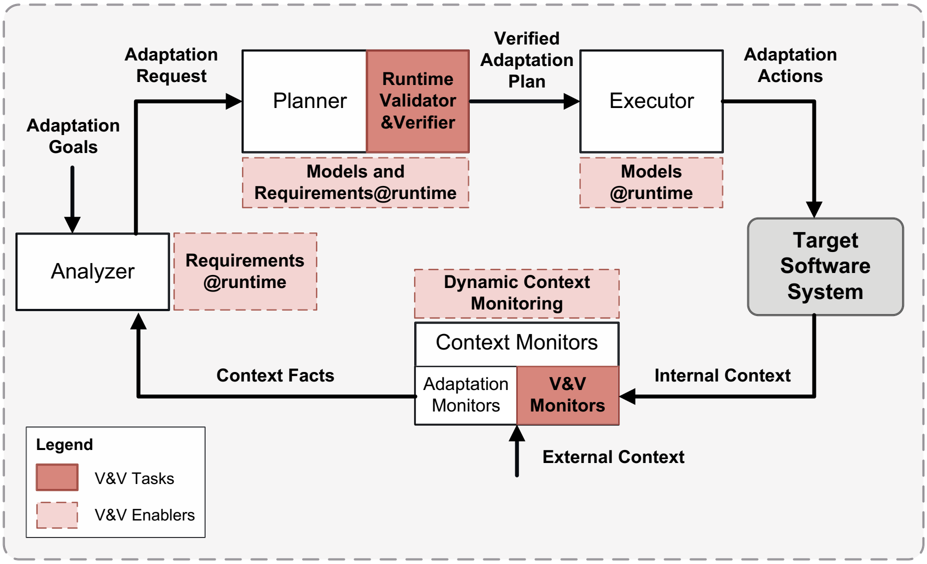


Fig. 2. Runtime V&V tasks made explicit as common elements in the engineering of self-adaptive software systems. Dashed boxes represent runtime V&V enablers.

verifiers can have access to the requirements and properties defined as adaptation goals that must be ensured by the adaptation process. Then, we use models at runtime to represent, the software architecture of the e-commerce platform to be adapted, the adaptation strategies, and the context monitoring strategies. Throughout the adaptation process, planners, runtime validators and verifiers, and executors use service component architecture (SCA) models to manipulate and adapt the system's software architecture, as well as to verify properties such as the SCA structural conformance, as realized in [30]. Similarly, we use models at runtime to represent the information gathered by context monitors as contextual RDF graphs. We exploit this form of representing context information to characterize the e-commerce platform state with respect to its viability zone, and perform inferences on this information to ensure SLAs at runtime [11].

4.1 Runtime V&V Tasks

We identify two particular elements in the adaptation loop that initiate runtime V&V tasks: *runtime validators & verifiers* (associated to the `Planner` element), and *V&V monitors* (associated to `Context Monitors` elements).

The Runtime Validator & Verifier. The responsibility of the *runtime validator & verifier* elements is to verify each of the outputs (i.e., adaptation plans) produced by the adaptation planner with respect to the properties of interest. The instrumentation of an adaptation plan on a given system execution state implies a change of the system state. Thus, the verification of these properties can be performed before or after instrumenting the plan.

In the case of our application example, concerning the SCA structural conformance property, the produced reconfiguration plans modify the target system's software architecture to obtain a new software structure to satisfy the agreed SLOs. To prevent execution failures, these plans must be verified, *before* instrumenting them, in such a way that the resulting structures satisfy the SCA integrity constraints (e.g., components, connectors, wires, and bindings). However, if the adaptation plan is for affecting the performance SLO, the corresponding verification should be performed *after* its instrumentation, with the new system's performance measurements. Moreover, on the new target system structure, partial and incremental verification could be performed also *in advance* on the most probable states that are immediately adjacent to the one generated by the adaptation plan. These states could be computed with statistical approaches such as the proposed in [27]. In addition, similar verification could be performed on the controller algorithm, if this is object of adaptation, such as in self-tuning control approaches [31, 32].

Nonetheless, different performance and synchronization issues between the executor and the runtime validator & verifier elements may appear. An example of this occurs when considering the previously introduced *in advance* partial and incremental V&V of the structural conformance property. In this case, the idea is to perform V&V not only on the state produced by the adaptation plan, but also on the most probable states that can immediately follow it, as a result of further adaptation processes. Thus, runtime V&V elements could verify the property of interest on these states either *at the same time*, or *after* instrumenting the plan to reach the produced state. In other words, if the function computing the next most probable states is correct, the system structure to be obtained with the produced adaptation plan had to be verified in the previous adaptation. Alternatively, the execution of this "advance" runtime verification can be delayed, and even scheduled for later execution, for instance if this verification is highly time-consuming and can compromise the performance SLO of the target system.

Finally, in those cases in which the system state is represented and maintained explicitly (i.e., having a stateful representation via, for example, reflection or models at runtime), and the system is modified by an adaptation, this explicit state has to be transformed or updated accordingly. This is especially critical if the state is maintained in a volatile data structure, whose layout changes when the system is reconfigured. That is, if system information is represented in one form in its state variables, and then the system is reconfigured such that this information is represented in a different form, then the old values from the old structure must be mapped into new values in the new structure. Moreover, for the system operation to continue safely and uninterruptedly, it is crucial to (i) make the new state (information and structure) persistent (e.g., for recovery purposes); and (ii) be able to initialize the new system with the old information mapped into the new state. Hence, in these cases V&V must be performed not only on the adaptation process, but also on the state-mapping from the old structure to the new one.

The V&V Monitors. *V&V monitors* are responsible for monitoring and enforcing the V&V tasks performed by the runtime validator & verifier elements. Referring to the V&V tasks assigned to the runtime validator & verifier elements in the example of the previous subsection, we could use the V&V monitors to perform the aforementioned "advance" runtime verification. As outlined in the previous subsection, this is a verification task that can be scheduled by the *runtime validator & verifier* elements for later execution, to be performed on the most probable states to the current one in execution.

Assurance of Runtime V&V Tasks. Derived from the previous discussion, we identify the following questions, which pose additional challenges for ensuring the effectiveness of V&V tasks.

What if V&V fails or provides a negative answer? To prevent the target system from reaching inconsistent states and avoid catastrophic situations, one first strategy is to guarantee the atomicity property in the adaptation process, as defined in [5, 33]. That is, to guarantee that the adaptation process is an atomic operation that finishes and successfully modifies the target system, or it fails and the target system is left unmodified in its previous safe state. The verification of the atomicity and termination properties is a challenging problem, given that they should be guaranteed internally by the planner, and possibly requiring interactions with the executor. The use of models at runtime for modeling the target system is crucial for guaranteeing these properties, for instance as realized by Tamura *et al.* [30].

How to validate "snapshots" and transitions between states without affecting the target system? V&V tasks must not affect the desired behaviour of the adaptive system. Therefore, we identify another kind of properties—properties of runtime V&V methods, intended to support the safe integration of traditional V&V techniques and mechanisms into the adaptation loop. These properties include *sensitivity, isolation, incrementality, and composability.*

As stated by González *et al.*, sensitivity and isolation refer to the level of runtime validation that a particular SAS system can support [34]. On the one hand, sensibility defines the degree to which V&V tasks (e.g., runtime testing operations) interfere with the running target system. That is, the degree to which runtime V&V may affect the satisfaction of system requirements and adaptation goals. Instances of factors that can affect runtime test sensitivity are (i) component state—not only because runtime validation tasks are influenced by the actual state of the system, but also because the state of the system can be altered as a result of V&V operations; (ii) component interactions—as the runtime testability of a component may depend on the testability of the components it interacts with; (iii) resource limitations—because runtime V&V may affect non-functional requirements on the target system, such as performance at undesirable levels; and (iv) availability—as runtime validation can be performed depending on whether testing tasks require exclusive usage of components with high availability requirements.

On the other hand, they also define isolation as the means to counteract runtime test sensitivity. Techniques for implementing test isolation include (i) state separation (e.g., blocking the component operation while testing takes place or performing testing on cloned components); (ii) interaction separation (e.g., blocking component interactions that may be propagated due to results of test invocations); (iii) resource monitoring (e.g., indicating that testing must be postponed due to resource unavailability); and (iv) scheduling (e.g., planning V&V executions when the target system and involved components are less used).

4.2 Runtime V&V Enablers

Runtime V&V techniques for SAS systems require special support to deal with the dynamic nature of this kind of systems in the assurance of adaptation goals. We classify this support in three main categories, as follows:

i. Enablers for the management of adaptation properties and requirements at runtime;
ii. Enablers for the exploitation of models at runtime; and
iii. Enablers for dynamic context monitoring.

Clearly, these categories correspond to challenges of the *Models@runtime* [35] and *Requirements@runtime* [36] communities, rather than research challenges of

V&V communities. Nevertheless, given that runtime V&V tasks for SAS rely on this support, with this categorization we aim to provide valuable guidance, not only for V&V researchers to understand the support that runtime V&V for SAS requires, but also for SEAMS-related researchers to visualize how could they attack runtime V&V challenges.

Requirements and Adaptation Properties at Runtime. The first category of support required for runtime V&V concerns the specification of *what* must be validated and verified. That is, the specification of the adaptation properties and system requirements the adaptation process must guarantee. In either case, V&V methods and techniques must determine whether the software product satisfies its requirements, especially after performing adaptation operations. These requirements and properties, expressed using different notations and formalisms, constitute the actual reference specifications for V&V tasks to accomplish their mission. Thus, requirements and adaptation goals must be available as machine-processable specifications (cf. Requirements@runtime in Fig. 2) to be used by adaptation analyzers, monitors, validators and verifiers. Furthermore, to minimize the impact of runtime V&V tasks on the adaptive system, support for tracing changes on requirements and properties is also required to identify what to validate and verify incrementally. Manipulating requirements and adaptation properties during the adaptive system execution poses interesting research questions such as the ones being addressed by the *Requirements@run.time* research community [36].

Models at Runtime. Having machine-processable models at runtime of the target system provides adaptation controllers, monitors, validators and verifiers with up-to-date structural and behavioral representations of the target system, and their relationships with adaptation properties and goals. Recalling our application example, after the renegotiation of the performance SLA (cf. Use Case 2 in Sect. 2), the runtime representations of the system and its requirements must change accordingly. That is, a new requirement is added to the context-driven SLA specification, as well as the corresponding monitoring strategy, using a contextual RDF graph. As a result, not only adaptation components, but also V&V tasks and monitors will have up-to-date representations of the new goals that must be ensured, and the corresponding context entities to be monitored.

Classifications of models at runtime vary from coarse-grained to fine-grained models, from structural to behavioural models, from dynamic to static models [35]. In this endeavor, researchers from the *Models@run.time* community are tackling important challenges [37, 38]. An instance of these challenges is model evolution, which concerns with the management of changes in models over time.

Model Evolution. Having an explicit representation of the target system, the properties to be preserved, and the relationships between these properties and adaptation mechanisms is critical for the assessment of SAS systems at runtime. At design time, models provide a meta-level representation of these concerns. At execution time, instances of these design time models, models at runtime, provide up-to-date representations of the system to V&V operations. These online representations support decision making on the preservation of desired properties. Since SAS systems are continuously changing, the effectiveness of runtime V&V tasks depends on the timely coherence between the actual system state and its runtime models. Model evolution support is therefore required to preserve the coherence of runtime models with respect to the system and its environment.

Model evolution for SAS systems can borrow relevant ideas from control-based approaches. These approaches include model reference adaptive control (MRAC) and model identification adaptive control (MIAC) [31, 39]. MRAC and MIAC not only separate adaptation models from adaptation controllers, but also V&V models from V&V tasks. As illustrated in Fig. 3, MRAC and MIAC enable a basic level of model evolution by modifying the adaptation and V&V models at runtime. The main difference between MRAC and MIAC, from the perspective of runtime V&V, is that in MRAC changes in models are controlled by users, whereas in MIAC changes in models are managed by executors as defined by runtime validators and verifiers. Changes in models (cf. label ChM: Change model) may cause changes not only in adaptation controllers, but also in V&V tasks (cf. labels SCh: Send changes, and AC: Adapt controllers). Therefore, runtime support is required to adapt runtime validators and verifiers accordingly (cf. label AC-VV). Changes in V&V models could be triggered by adaptation mechanisms. In any case, these changes must be subject to V&V operations. From a software engineering perspective, the probabilistic approach to model synchronization proposed by Epifani *et al.* constitutes a good MIAC approach to model evolution [32].

Different model evolution mechanisms can be applied depending on the modeling technique used. For example, to synchronize UML models with corresponding systems, the model-driven engineering community provides model transformation techniques applicable at runtime [40]. In systems relying on probabilistic models, synchronization of models is realized by changing the model's parameters at runtime. One key challenge in probabilistic models is to synchronize the measured probabilities with the probabilities used in the model [32].

Dynamic Context Monitoring. The third category of V&V support corresponds to runtime context monitoring. Context monitoring is crucial to optimize the assessment of dynamic software systems as the effectiveness of V&V methods is highly dependent on the information provided by context sources [41]. These context information sources must be consistent with the actual system adaptation properties and requirements. Thus, for V&V tasks to succeed in the assessment of a SAS system, it must understand the situations of relevant context entities and their implications for the preservation of system properties and requirements, even when these requirements and properties vary over time.

Context is any information useful to characterize the state of individual entities and the relationships among them. An entity is any subject that can affect the behaviour of the system and/or its interaction with the user. Context information must be modeled in such a way that it can be pre-processed after its acquisition from the environment, classified according to the corresponding domain, handled to be provisioned based on the system's requirements, and

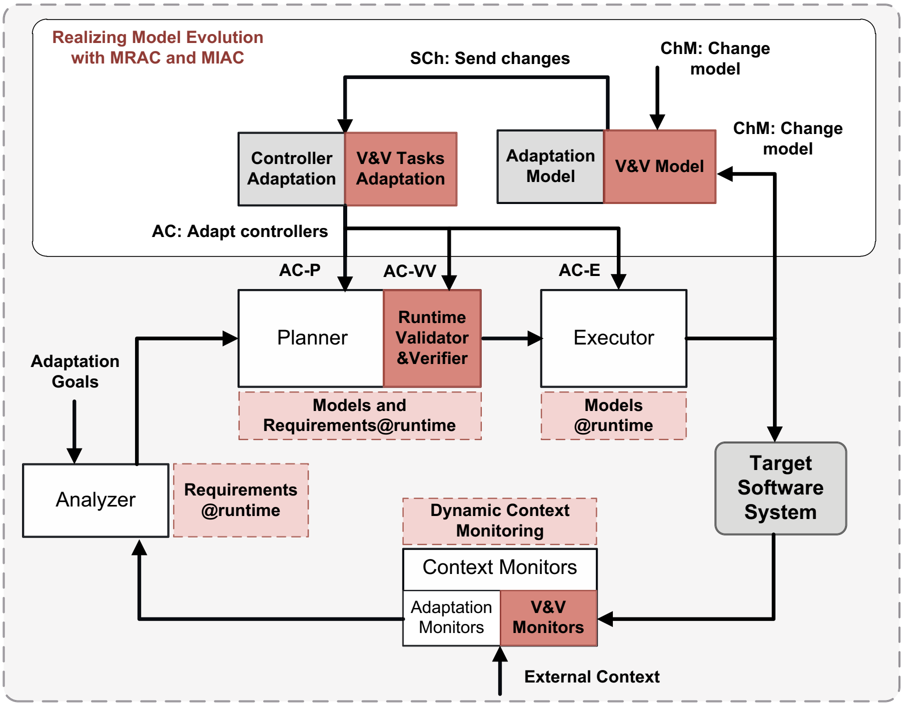


Fig. 3. MRAC and MIAC [31, 39] reinterpreted to realize model evolution in self-adaptation with explicit V&V tasks

maintained to support its dynamic evolution [42]. Based on this definition, and from the perspective of runtime V&V, runtime monitoring must support context representation and monitoring to characterize the system's state with respect to its viability zone, taking into account the dynamic nature of viability zones. Regarding context representation, operational specifications of context information must be able to represent semantic dependencies among properties and requirements to be satisfied, V&V strategies, and the environmental situations that have impact on the system behaviour and the assessment tasks. Hence, an important challenge refers to context representation such that these specifications can adapt dynamically, according to changes in V&V concerns. With respect to context management, several challenges arise from the perspective of the context information lifecycle, that is context acquisition, handling, provisioning, maintenance, and disposal [42]. One of these challenges is the instrumentation of monitoring infrastructures with dynamic capabilities to deploy new monitoring strategies at runtime according to changes in V&V concerns (e.g., to deploy new sensors or reasoning strategies based on changes in adaptation goals and properties dynamically [11, 25, 43]). In our application example, the monitoring infrastructure must be adapted at runtime to deploy new context sensors and monitoring conditions provided with the new response time SLO that resulted from contract re-negotiation (cf. Use Case 2 in Sect. 2).

Runtime monitoring could help alleviate issues concerning the application of traditional V&V techniques at runtime. An instance of such issues is the state explosion problem inherent in model checking techniques. In adaptive software systems, the uncertainty of the execution environment, the dynamic nature of system requirements, and the continuous adaptation of systems exacerbate the problem. From this perspective, we hypothesize that if we are able to characterize the current state of a system at a specific time during its execution, the number of system states to be checked could be significantly smaller. At design time many variables are free or not bounded, thus all of their possible significant values must be checked. In contrast, at runtime, variables are bound using the actual system state and the situation of relevant environmental (internal and external) entities. In other words, the number of possible states for the system to maintain within its viability zone is considerably reduced by the current and next most probable context situations. This is precisely where *context monitoring* plays a crucial role in the assessment of self-adaptive software systems. Nevertheless, due to the uncertainty inherent in dynamic software systems, it is infeasible to specify context requirements in advance exhaustively. Moreover, since context is evolving over time, monitoring requirements—entities to be monitored and monitoring conditions—are also continuously evolving. Therefore, the application of traditional V&V techniques to the assessment of self-adaptive systems at runtime depends on the dynamic capabilities of the runtime monitoring instrumentation.

5 Conclusions

In this roadmap chapter we (i) discussed key challenges for the development of certifiable runtime V&V methods that can certify adaptation mechanisms in the achievement of their adaptation goals; and (ii) presented how to make explicit and integrate runtime V&V methods as concrete tasks to be performed by elements of the adaptation process. Certifiable V&V methods and tools are critical for the success of autonomous, autonomic, smart, self-adaptive and self-managing systems.

We defined control science as a systematic way to study certifiable runtime V&V methods and tools to allow humans to trust decisions made by self-adaptive systems. For the first contribution, we analyzed critical differences between SAS systems and non-adaptive ones. From these differences, we identified and discussed key factors and challenges to consider when tailoring existing V&V methods, or developing new ones, to be applied at runtime in SAS systems. For the second, we discussed and illustrated different possibilities to integrate these V&V methods as responsibilities for the adaptation process elements. To enable this integration, we analyzed how to exploit some of the foundational ideas developed by SEAMS-related research communities to support the application of V&V methods at runtime.

While self-adaptation confers obvious benefits to systems with high levels of adaptability and autonomy, its wide adoption by industry is still limited due to a lack of self-applicable validation and verification methods at runtime. The positive impact that self-adaptive software can have on our society is potentially huge. Nevertheless, without the necessary and sufficient trustworthy certification methods the negative impact can be potentially huge too. Consequently, research and development in runtime assurance techniques is critical to guarantee that adaptation mechanisms will not cause the target system to produce undesired, nor catastrophic results.

Therefore, our motivation for this chapter was to provide researchers with a vision of open challenges in V&V for SAS systems, and discuss opportunities not only for proposing new runtime V&V techniques, but also for building on top of existing ones. In addition, our proposal for making V&V tasks explicit in the adaptation loop provides solid starting points for V&V researchers from other communities to deploy different techniques and methods for improving the trustworthiness of self-adaptive and self-managing systems. For this, we analyzed runtime assessment concerns from the perspective of *when* in the adaptation process, and in which of the two parts of an adaptive system (i.e., the *where*)—the target system or the adaptation mechanism—the V&V tasks must be implemented and performed.

The questions discussed in this chapter have uncovered key research problems that require collaborative efforts among different software engineering research communities. In particular, models at runtime, requirements at runtime, validation and verification, and context monitoring have in the assessment of adaptive software a unique opportunity to advance the state-of-the-art software engineering for self-adaptive systems. With our contributions in this chapter we aim to provide researchers from various runtime V&V communities with research avenues that can shape the development of certifiable assurance techniques, as required for the engineering of trustworthy SAS systems.

Acknowledgments. This chapter was motivated by stimulating discussions during Dagstuhl Seminar 10431 on Software Engineering for Self-Adaptive Systems at Schloss Dagstuhl in October 2010. This work was funded in part by the National Sciences and Engineering Research Council (NSERC) of Canada under the Strategic Networks Grants Program (NETGP 397724-10) and Collaborative Research and Development program (CRDPJ 320529-04 and CRDPJ 356154-07), IBM Corporation, CA Inc., Icesi University (Cali, Colombia), and Ministry of Higher Education and Research of Nord-Pas de Calais Regional Council and FEDER under Contrat de Projets Etat Region (CPER) 2007-2013.

References

1. IEEE: 1012-1998: IEEE Standard for Software Verification and Validation. Technical report, Institute of Electrical and Electronics Engineers (2005)
2. IEEE: Industry Implementation of International Standard ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes. Technical report, IEEE (1996)
3. Bourque, P., Dupuis, R.: Guide to the Software Engineering Body of Knowledge (SWEBOK). IEEE Computer Society (2005)

4. Salehie, M., Tahvildari, L.: Self-Adaptive Software: Landscape and Research Challenges. ACM Transactions on Autonomous and Adaptive Systems 4, 14:1–14:42 (2009)
5. 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 (SEAMS 2011), pp. 80–89. ACM, New York (2011)
6. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
7. Schafer, W., Wehrheim, H.: The Challenges of Building Advanced Mechatronic Systems. In: 2007 Future of Software Engineering (FOSE 2007), pp. 72–84. IEEE Computer Society, Washington, DC (2007)
8. Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., Koschke, R.: A Systematic Survey of Program Comprehension through Dynamic Analysis. IEEE Transactions on Software Engineering (TSE) 35, 684–702 (2009)
9. Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D.M.: Feedback Control of Computing Systems. John Wiley & Sons (2004)
10. Dahm, W.J.A.: Technology Horizons a Vision for Air Force Science & Technology During 2010-2030. Technical report, U.S. Air Force (2010)
11. Villegas, N.M., Müller, H.A., Tamura, G.: Optimizing Run-Time SOA Governance through Context-Driven SLAs and Dynamic Monitoring. In: 2011 IEEE International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA 2011), pp. 1–10. IEEE (2011)
12. Beisiegel, M., Blohm, H., Booz, D., Edwards, M., Hurley, O., et al.: Service Component Architecture, Assembly Model Specification. Specification Version 1.0, Open Service Oriented Architecture (OSOA) Collaboration (2007)
13. Thayer, R.H., Bailin, S.C., Dorfman, M.: Software Requirements Engineering, 2nd edn. IEEE Computer Society Press, Los Alamitos (1997)
14. Dorfman, M.: System and Software Requirements Engineering, pp. 7–22. IEEE Computer Society Press Tutorial, IEEE Computer Society Press (1990)
15. Pezzè, M., Young, M.: Software Test and Analysis: Process, Principles and Techniques. John Wiley and Sons, Hoboken (2008)
16. Gat, E.: Autonomy Software Verification and Validation might not be as Hard as it Seems (AeroConf 2004). In: 2004 IEEE Aerospace Conference, pp. 3123–3128 (2004)
17. Bucchiarone, A., Pelliccione, P., Vattani, C., Runge, O.: Self-Repairing Systems Modeling and Verification Using AGG. In: 8th IEEE/IFIP Joint Working International Conference on Software Architecture (WICSA) and 3rd European Conference on Software Engineering (ECSA), pp. 181–190. IEEE (2009)
18. Bose, P., Quilling, M.: Model-Based Analysis of Autonomous Self-Adaptive Cooperating Robots. In: 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2008), pp. 57–63. IEEE Computer Society, Washington, DC (2008)

19. Murray, R.M. (ed.): Control in an Information Rich World: Report of the Panel on Future Directions in Control, Dynamics, and Systems. Society for Industrial and Applied Mathematics, Philadelphia (2003)
20. Baresi, L., Ghezzi, C.: The Disappearing Boundary between Development-time and Run-time. In: FSE/SDP Workshop on Future of Software Engineering Research (FoSER 2010), pp. 17–22. ACM, New York (2010)
21. Aubin, J., Bayen, A., Saint-Pierre, P.: Viability Theory: New Directions. Springer, Heidelberg (2011)
22. Balasubramanian, S., Desmarais, R., Müller, H.A., Stege, U., Venkatesh, S.: Characterizing Problems for Realizing Policies in Self-Adaptive and Self-Managing Systems. In: 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011), pp. 70–79. ACM, New York (2011)
23. Jacklin, S.A., Lowry, M.R., Schumann, J.M., Gupta, P.P., Bosworth, J.T., Zavala, E., Kelly, J.W.: Verification, Validation, and Certification Challenges for Adaptive Flight-Critical Control System Software. In: American Institute of Aeronautics and Astronautics AIAA Guidance Navigation and Control Conference and Exhibit. American Institute of Aeronautics and Astronautics, pp. 1–10 (2004)
24. Crum, V.W., Buffington, J.M., Tallant, G.S., Krogh, B., Plaisted, C., Prasanth, R., Bose, P., Johnson, T.: Verification & Validation of Intelligent and Adaptive Control Systems. In: IEEE Aerospace Conference (AeroConf. 2004), pp. 68–77. IEEE Computer Society (2004)
25. Villegas, N.M., Müller, H.A., Muñoz, J.C., Lau, A., Ng, J., Brealey, C.: A Dynamic Context Management Infrastructure for Supporting User-driven Web Integration in the Personal Web. In: 2011 Conference of the Center for Advanced Studies on Collaborative Research (CASCON 2011), pp. 200–214. IBM Corp, Markham (2011)
26. Murray, R.M., Åström, K.J., Boyd, S.P., Brockett, R.W., Stein, G.: Future Directions in Control in an Information Rich World. IEEE Control Systems 23, 20–33 (2003)
27. Schumann, J., Gupta, P.: Bayesian Verification & Validation Tools for Adaptive Systems: Report on Principle of Operation and Prototypical Implementation of Bayesian Envelope Tool for Neural Networks. Technical report, National Aeronautics and Space Administration, NASA (2006)
28. Elkhodary, A., Esfahani, N., Malek, S.: FUSION: A Framework for Engineering Self-Tuning Self-Adaptive Software Systems. In: 18th ACM International Symposium on Foundations of Software Engineering, FSE 2010, pp. 7–16. ACM, (2010)
29. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. IEEE Computer 36(1), 41–50 (2003)
30. Tamura, G., Casallas, R., Cleve, A., Duchien, L.: QoS Contract-Aware Reconfiguration of Component Architectures Using E-Graphs. In: Barbosa, L.S. (ed.) FACS 2010. LNCS, vol. 6921, pp. 34–52. Springer, Heidelberg (2010)
31. Dumont, G., Huzmezan, M.: Concepts, Methods and Techniques in Adaptive Control. In: 2002 IEEE American Control Conference (ACC 2002), Anchorage, AK, USA, vol. 2, pp. 1137–1150 (2002)
32. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model Evolution by Run-Time Parameter Adaptation. In: 31st International Conference on Software Engineering (ICSE 2009), pp. 111–121. IEEE (2009)
33. Léger, M., Ledoux, T., Coupaye, T.: Reliable Dynamic Reconfigurations in a Reflective Component Model. In: Grunske, L., Reussner, R., Plasil, F. (eds.) CBSE 2010. LNCS, vol. 6092, pp. 74–92. Springer, Heidelberg (2010)

34. González, A., Piel, E., Gross, H.G.: A Model for the Measurement of the Runtime Testability of Component-Based Systems. In: 2009 International Conference on Software Testing Verification and Validation Workshops (ICSTW), pp. 19–28. IEEE (2009)
35. Bencomo, N., Blair, G., France, R., Muñoz, F., Jeanneret, C.: 4th International Workshop on Models@run.time. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 119–123. Springer, Heidelberg (2010)
36. Sawyer, P., Bencomo, N., Whittle, J., Letier, E., Finkelstein, A.: Requirements-Aware Systems. A Research Agenda for RE For Self-Adaptive Systems. In: 18th International Requirements Engineering Conference (RE 2010), pp. 95–103. IEEE (2010)
37. Blair, G., Bencomo, N., France, R.: Models@run.time. IEEE Computer 42, 22–27 (2009)
38. France, R., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. In: 2007 Future of Software Engineering (FOSE 2007). IEEE Computer Society (2007)
39. Müller, H.A., Kienle, H.M., Stege, U.: Autonomic Computing Now You See It, Now You Don't. In: De Lucia, A., Ferrucci, F. (eds.) ISSSE 2006-2008. LNCS, vol. 5413, pp. 32–54. Springer, Heidelberg (2009)
40. Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Incremental Model Synchronization for Efficient Run-Time Monitoring. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 124–139. Springer, Heidelberg (2010)
41. Goldsby, H., Cheng, B., Zhang, J.: AMOEBA-RT: Run-Time Verification of Adaptive Software. In: Giese, H. (ed.) MODELS 2008. LNCS, vol. 5002, pp. 212–224. Springer, Heidelberg (2008)
42. Villegas, N.M., Müller, H.A.: Managing Dynamic Context to Optimize Smart Interactions and Services. In: Chignell, M., Cordy, J., Ng, J., Yesha, Y. (eds.) The Smart Internet. LNCS, vol. 6400, pp. 289–318. Springer, Heidelberg (2010)
43. Villegas, N.M., Müller, H.A.: Context-driven Adaptive Monitoring for Supporting SOA Governance. In: 4th International Workshop on a Research Agenda for Maintenance and Evolution of Service-Oriented Systems (MESOA 2010). CMU/SEI-2011-SR-008, Pittsburgh: Carnegie Mellon University (2011)

Awareness Requirements*

Vítor E. Silva Souza[1], Alexei Lapouchnian[1],
William N. Robinson[2], and John Mylopoulos[1]

[1] Department of Inf. Engineering and Computer Science, University of Trento, Italy
{vitorsouza,lapouchnian,jm}@disi.unitn.it
[2] Department of Computer Information Systems, Georgia State University, USA
wrobinson@gsu.edu

Abstract. The functional specification of any software system operationalizes stakeholder requirements. In this paper we focus on a class of requirements that lead to feedback loop operationalizations. These *Awareness Requirements* talk about the runtime success/failure of other requirements and domain assumptions. Our proposal includes a language for expressing awareness requirements, as well as techniques for elicitation and implementation based on the EEAT requirements monitoring framework.

1 Introduction

There is much and growing interest in software systems that can adapt to changes in their environment or their requirements in order to continue to fulfill their mandate. Such adaptive systems usually consist of a system proper that delivers a required functionality, along with a monitor-analyze-plan-execute (MAPE [18]) feedback loop that operationalizes the system's adaptability mechanisms. Indications for this growing interest can be found in recent workshops and conferences on topics such as adaptive, autonomic and autonomous software (e.g., [7,23,14]).

We are interested in studying the *requirements* that lead to this feedback loop functionality. In other words, if feedback loops constitute an (architectural) solution, what is the requirements problem this solution is intended to solve? The nucleus of an answer to this question can be gleamed from any description of feedback loops: "... the objective ... is to make some output, say y, behave in a desired way by manipulating some input, say u ..." [10]. Suppose then that we have a requirement r = "supply customer with goods upon request" and let s be a system operationalizing r. The "desired way" of the above quote for s is that it *always* fulfills r, i.e., every time there is a customer request the system meets it successfully (here, the notion of "success" depends on the type of system: for software systems, it means completing the transaction without errors or exceptions, whereas for socio-technical systems "success" could involve the participation of human actors, e.g., goods are properly delivered to the customer). This means

* This is an extended version of the paper titled "Awareness Requirements for Adaptive Systems" published in the proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS "11), pages 60–69. ACM, 2011.

R. de Lemos et al. (Eds.): Self-Adaptive Systems, LNCS 7475, pp. 133–161, 2013.

that the system somehow manages to deliver its functionality under all circumstances (e.g., even when one of the requested items is not available). Such a requirement can be expressed, roughly, as `r1` = “Every instance of requirement `r` succeeds”. And, of course, an obvious way to operationalize `r1` is to add to the architecture of `s` a feedback loop that monitors if system responses to requests are being met, and takes corrective action if they are not. We can generalize on this: we could require that `s` succeeds more than 95% of the time over any one-month period, or that the average time it takes to supply a customer over any one week period is no more than 2 days. The common thread in all these examples is that they define requirements about the run-time success/failure/quality-of-service of other requirements. We call these *self-awareness requirements.*

A related class of requirements is concerned with the truth / falsity of domain assumptions. For our example, we may have designed our customer supply system on the domain assumption `d` = “suppliers for items we distribute are always open”. Accordingly, if supplier availability is an issue for our system, we may want to add yet another requirement `r2` = “`d` will not fail more than 2% of the time during any 1-month period”. This is also an awareness requirement, but it is concerned with the truth/falsity of a domain assumption.

The objective of this paper is to study Awareness Requirements (hereafter referred to as *AwReqs*), which are characterized syntactically as requirements that refer to other requirements or domain assumptions and their success or failure at runtime. *AwReqs* are represented in an existing language and can be directly monitored by a requirements monitoring framework. Although the technical contribution of this paper is focused on the definition and study of *AwReqs* and their monitoring at runtime, we do provide a discussion on how to go from *AwReqs* to adaptive systems, giving an overview of subsequent steps in this process.

Awareness is a topic of great importance within both Computer and Cognitive Sciences. In Philosophy, awareness plays an important role in several theories of consciousness. In fact, the distinction between self-awareness and contextual requirements seems to correspond to the distinction some theorists draw between higher-order awareness (the awareness we have of our own mental states) and first-order awareness (the awareness we have of the environment) [29]. In Psychology, consciousness has been studied as “self-referential behavior”. Closer to home, awareness is a major design issue in Human-Computer Interaction (HCI) and Computer-Supported Cooperative Work (CSCW). The concept in various forms is also of interest in the design of software systems (security / process / context / location / ... awareness).

As part of our proposal’s evaluation, which we detail in section 5, we have analyzed, designed and developed a simulation of a real-world system: an Ambulance Dispatch System (ADS), whose requirements have been documented by students of the University of Texas at Dallas [28]. We will use this application as running example throughout this paper.

The rest of the paper is structured as follows. Section 2 presents the research baseline; section 3 introduces *AwReqs* and talks about their elicitation; section 4 discusses their specification; section 5 talks about *AwReqs* monitoring implementation and presents evaluation results from experiments with our proposal; section 6 summarizes related work; section 7 discusses the role of *AwReqs* in a systematic process for the development of adaptive systems based on feedback loops; finally, section 8 concludes the paper.

2 Baseline

This section introduces background research used in subsequent sections of this paper: Goal-Oriented Requirements Engineering (§2.1), feedback loops (§2.2) and requirements monitoring (§2.3).

2.1 Goal-Oriented Requirements Engineering

Our proposal is based on Goal-oriented Requirements Engineering (GORE). GORE is founded on the premise that requirements are stakeholder *goals* to be fulfilled by the system-to-be along with other actors. Goals are elicited from stakeholders and are analyzed by asking "why" and "how" questions [8]. Such analysis leads to goal models which are partially ordered graphs with stakeholder requirements as roots and more refined goals lower down. Our version of goal models is based loosely on $i^\star$ strategic rationale models [37]. Figure 1 shows a goal model for an Ambulance Dispatch System (ADS).

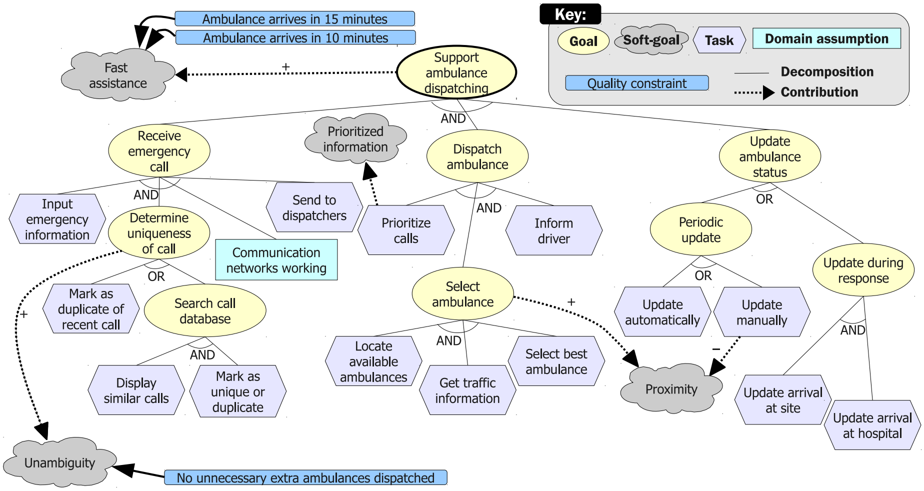


Fig. 1. Example goal model for an Ambulance Dispatch System

In our example, the main goal of the system is to support ambulance dispatching. Goals can be AND/OR refined. An AND-refinement means that in order to accomplish the parent goal, all sub-goals must be satisfied, while for an OR-refinement, only one of the sub-goals has to be attained. For example, to receive an emergency call, one has to input its information, determine its uniqueness (have there been other calls for the same emergency?) and send it to dispatchers, all on the assumption that "Communication networks [are] working"[1]. On the other hand, periodic update of an ambulance's status can be performed either automatically or manually.

Goals are refined until they reach a level of granularity where there are tasks an actor (human or system) can perform to fulfill them. In the figure, goals are represented as ovals and tasks as hexagons. Note that we represent AND/OR *refinement* relations, avoiding the term *decomposition* as it usually carries a part-whole semantic which would constrain its use among elements of the same kind[2] (i.e., goal to goal, task to task, etc.). A refinement relation, on the other hand, can be applied between a goal and a task or a goal and a domain assumption and indicate how to satisfy the parent element: the goal is satisfied if all (AND) or any (OR) of its children are satisfied. In their turns, tasks are satisfied if they are executed successfully and domain assumptions are satisfied if they hold (the affirmation is true) while the user is pursuing its parent goal.

Softgoals are special types of goals that do not have clear-cut satisfaction criteria. In our example, stakeholders would like ambulance dispatching to be fast, dispatched calls to be unambiguous and prioritized, and selected ambulances to be as close as possible to the emergency site. Softgoal satisfaction can be estimated through qualitative contribution links that propagate satisfaction or denial and have four levels of contribution: break (- -), hurt (-), help (+) and make (++). E.g., selecting an ambulance using the software system contributes positively to the proximity of the ambulance to the emergency site, while using manual ambulance status update, instead of automatic, contributes negatively to the same criterion. Contributions may exist between any two goals (including hard goals).

Softgoals are obvious starting points for modeling non-functional requirements. To make use of them in design, however, they need to be refined to measurable constraints on the system-to-be. These are quality constraints (QCs), which are perceivable and measurable entities that inhere in other entities [17]. In our example, unambiguity is measured by the number of times two ambulances are dispatched to the same location, while fast assistance is refined into two QCs: ambulances arriving within 10 or 15 minutes to the emergency site.

Finally, domain assumptions (DAs) indicate states of the world that we assume to be true in order for the system to work. For example, we assume that communication networks (telephone, Internet, etc.) are available and functional. If this assumption were to be false, its parent goal ("Receive emergency call") would not be satisfied.

[1] These requirements are for illustrative purposes and, thus, are quite simple. Real-world systems would probably have multiple domain assumptions, one for each level of communication service, or even have assumptions parameterized by control variables that can be tuned at runtime — see §7.1 for a discussion on control variables.

[2] One could argue that it makes no sense to consider a task or a domain assumption a part of a goal. In effect, we have received such criticism in the past, in more than one occasion.

2.2 Feedback Loops

The recent growth of software systems in size and complexity made it increasingly infeasible to maintain them manually. This led to the development of a new class of self-adaptive systems, which are capable of changing their behavior at runtime due to failures as well as in response to changes in themselves, their environment, or their requirements. While attempts at adaptive systems have been made in various areas of computing, Brun et al. [6] argue for systematic software engineering approaches for developing self-adaptive systems based on the ideas from control engineering [15] with focus on explicitly specified feedback loops. Feedback loops provide a generic mechanism for self-adaptation. To realize self-adaptive behavior, systems typically employ a number of feedback controllers, possibly organized into controller hierarchies.

The main idea of feedback control is to use measurements of a system's outputs to achieve externally specified goals [15]. The objective of a feedback loop is usually to maintain properties of the system's output at or close to its reference input. The measured output of the system is evaluated against the reference input and the control error is produced. Based on the control error, the controller decides how to adjust the system's control input (parameters that affect the system) to bring its output to the desired value. To do that, the controller needs to possess a model of the system. In addition, a disturbance may influence the way control input affects output. Sensor noise may be present as well. This view of feedback loops does not concentrate on the activities within the controller itself. That is the emphasis of another model of a feedback loop, often called the autonomic control loop [9]. It focuses on the activities that realize feedback: monitoring, analysis, plan, execution — MAPE [18].

The common control objectives of feedback loops are regulatory control (making sure that the output is equal or near the reference input), disturbance rejection (ensuring that disturbances do not significantly affect the output), constrained optimization (obtaining the "best" value for the measured output) [15]. Control theory is concerned with developing control systems with properties such as stability (bounded input produces bounded output), accuracy (the output converges to the reference input), etc. While most of these guidelines are best suited for physical systems, many can be used for feedback control of software systems.

Using the ADS as an example, a feedback loop would: (1) monitor particular indicators of the system which are of interest to the stakeholders — e.g., the time it takes for ambulances to arrive at the location of the incidents; (2) compare the monitored values of these indicators with reference values specified in the requirements — e.g., QCs in the ADS goal model indicate ambulances should arrive in 10 or 15 minutes; and (3) if the monitored values do not satisfy the requirements, do something to fix the problem — e.g., increase the number of ambulances, change their locations around the city, etc. In this paper we propose *Awareness Requirements* as indicators to be monitored by the feedback loop, whereas the other steps of the loop in the context of our research are briefly discussed in section 7. Our view of adaptive systems as control systems has also been featured in a recently published position paper [34].

2.3 Requirements Monitoring

Monitoring is the first step in MAPE feedback loops and, as will be characterized in section 3, since *AwReqs* refer to the success/failure of other requirements, we will need to monitor requirements at runtime.

Therefore, we have based the monitoring component of our implementation on the requirements monitoring framework EEAT[3], formerly known as ReqMon [24]. EEAT, an Event Engineering and Analysis Toolkit, provides a programming interface (API) that simplifies temporal event reasoning. It defines a language to specify goals and can be used to compile monitors from the goal specification and evaluate goal fulfillment at runtime.

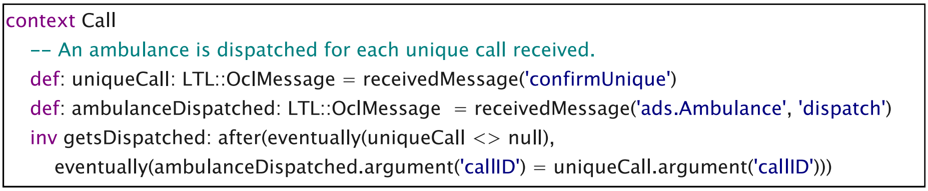
EEAT's architecture is presented in more detail along with our implementation in section 5. In it, requirements can be specified in a variant of the Object Constraints Language (OCL), called OCL$_{TM}$ — meaning OCL with Temporal Message logic [25]. OCL$_{TM}$ extends OCL 2.0 [2] with:

- Flake's approach to messages [12]: replaces the confusing ^ `message()`, ^^ `message()` syntax with `sentMessage`/`s`, `receivedMessage`/`s` attributes in class `OclAny`;
- Standard temporal operators: ○ (`next`), • (`prior`), ◊ (`eventually`), ♦ (`previously`), □ (`always`), ■ (`constantly`), $\mathcal{W}$ (`always` ... `unless`), $\mathcal{U}$ (`always` ... `until`);
- The scopes defined by Dwyer et al. [11]: `globally`, `before`, `after`, `between` and `after` ... `until`. Using the scope operators simplifies property specification;
- Patterns, also in Dwyer et al. [11]: `universal`, `absence`, `existence`, `bounded existence`, `response`, `precedence`, `chained precedence` and `chained response`;
- Timeouts associated with scopes: e.g. `after(Q, P, '3h')` indicates that `P` should be satisfied within three hours of the satisfaction of `Q`.

Figure 2 shows an example of OCL$_{TM}$ constraint on the ADS. The invariant `getsDispatched` determines that if a call receives the `confirmUnique` message, eventually an ambulance should get the message `dispatch` and both messages should refer to the same `callID` argument. Given an instrumented Java implementation of these objects and a program in which they exchange messages through method calls, EEAT is able to monitor and assert this invariant at runtime. In section 5, we describe in more detail how EEAT accomplishes this in the context of *AwReqs* monitoring.

Although in our proposal *AwReqs* can be expressed in any language that provides temporal constructs (e.g., LTL, CTL, etc.), examples of *AwReq* specifications in section 4 will be given using OCL$_{TM}$, which is also the language used for our proposal's validation, presented in section 5.

[3] `http://eeat.cis.gsu.edu:8080/`



```
context Call
   -- An ambulance is dispatched for each unique call received.
   def: uniqueCall: LTL::OclMessage = receivedMessage('confirmUnique')
   def: ambulanceDispatched: LTL::OclMessage  = receivedMessage('ads.Ambulance', 'dispatch')
   inv getsDispatched: after(eventually(uniqueCall <> null),
      eventually(ambulanceDispatched.argument('callID') = uniqueCall.argument('callID')))
```

Fig. 2. An example of OCL$_{TM}$ constraint

3 Awareness Requirements

As we have mentioned in section 1, feedback loops can provide adaptivity for a given system by introducing activities such as monitoring, analysis (diagnosis), planning and execution (of compensations) to the system proper. We are interested in modeling the requirements that lead to this feedback loop functionality. In control system terms (see §2.2), the reference input in this case is the system fulfilling its mandate (its requirements). Feedback loops, then, need to measure the actual output and compare it to the reference input, in other words, verify if requirements are being satisfied or not.

Furthermore, Berry et al. [4] defined the *envelope of adaptability* as the limit to which a system can adapt itself: “since for the foreseeable future, software is not able to think and be truly intelligent and creative, the extent to which a [system] can adapt is limited by the extent to which the adaptation analyst can anticipate the domain changes to be detected and the adaptations to be performed.”

In this context, to completely specify a system with adaptive characteristics, requirements for adaptation have to be included in the specifications. We propose a new kind of requirement, which we call Awareness Requirement, or *AwReq*, to fill this need. *AwReqs* promote feedback loops for adaptive systems to first-class citizens in Requirements Engineering.

In this section, we characterize *AwReqs* as requirements for feedback loops that implement adaptivity (§3.1); propose patterns to facilitate their elicitation, along with a way to represent them graphically in the goal model (§3.2); and discuss the elicitation of this new type of requirements (§3.3). We illustrate all of our ideas using our running example, the ADS (figure 1).

3.1 Characterization

AwReqs are requirements that talk about the run-time status of other requirements. Specifically, *AwReqs* talk about the states requirements can assume during their execution at runtime. Figure 3 shows these states which, in the context of our modeling framework, can be assumed by goals, tasks, DAs, QCs and *AwReqs* themselves. When an actor starts to pursue a requirement, its result is yet `Undecided`. Eventually, the requirement will either have `Succeeded`, or `Failed`. For goals and tasks, there is also a `Canceled` state.

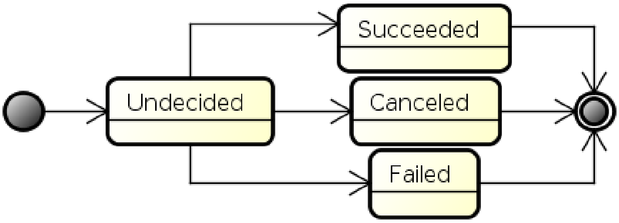


Fig. 3. States assumed by a requirement at runtime

Table 1. Examples of *AwReqs*, elicited in the context of the ADS

Id	**Description**	**Type**	**Pattern**
AR1	*Input emergency information* should never fail	–	`NeverFail(T-InputInfo)`
AR2	*Communications networks working* should have 99% success rate	Aggregate	`SuccessRate(D-CommNets Work, 99%)`
AR3	*Search call database* should have a 95% success rate over one week periods	Aggregate	`SuccessRate(G-Search CallDB, 95%, 7d)`
AR4	*Dispatch ambulance* should fail at most once a week	Aggregate	`MaxFailure(G-Dispatch Amb, 1, 7d)`
AR5	*Ambulance arrives in 10 minutes* should succeed 60% of the time, while *Ambulance arrives in 15 minutes* should succeed 80%, measured daily	Aggregate	`@daily SuccessRate( Q-Amb10min, 60%) and SuccessRate(Q-Amb15min, 80%)`
AR6	*Update automatically* should succeed 100 times more than the task *Update manually*	Aggregate	`ComparableSuccess( T-UpdAuto, T-UpdManual, 100)`
AR7	The success rate of *No unnecessary extra ambulances* for a month should not decrease, compared to the previous month, two times consecutively	Trend	`not TrendDecrease( Q-NoExtraAmb, 30d, 2)`
AR8	*Update arrival at site* should be successfully executed within 10 minutes of the successful execution of *Inform driver*, for the same emergency call	Delta	`ComparableDelta( T-UpdArrSite, T-InformDriver, time, 10m)`
AR9	*Mark as unique or duplicate* should be decided within 5 minutes	Delta	`StateDelta(T-MarkUnique, Undecided, *, 5m)`
AR10	`AR3` should have 75% success rate over one month periods	Meta	`SuccessRate(AR3, 75%, 30d)`
AR11	`AR5` should never fail	Meta	`NeverFail(AR5)`

Table 1 shows some of the *AwReqs* that were elicited during the analysis of the ADS. These examples illustrate the different types of *AwReqs*, which are discussed in the following paragraphs. Table 1 also indicates the pattern of each *AwReq* and we further elaborate on this matter on section 3.2.

The examples illustrate a number of types of *AwReq*. `AR1` shows the simplest form of *AwReq*: the requirement to which it refers should never fail. Considering a control system, the reference input is to fulfill the requirement. If the actual output is telling us the requirement has failed, the control system must act (compensate, reconcile — out of the scope of this proposal and briefly discussed in section 7) in order to bring the system back to an acceptable state. `AR1` considers every instance of the referred requirement. An instance of a task is created every time it is executed and the “never fail” constraint is to be checked for every such instance. Similarly, instances of a goal exist whenever the goal needs to be fulfilled, while DA and QC instances are created whenever their truth/falsity needs to be checked in the context of a goal fulfillment.

Inspired by the three modes of control of the proportional-integral-differential (PID) controller, a widely used feedback controller type [10], we propose three types of *AwReqs*: *Aggregate AwReqs* act like the integral component, which considers not only the current difference between the output and the reference input (the control error), but aggregates the errors of past measurements. *Delta AwReqs* were inspired by how proportional control sets its output proportional to the control error. *Trend AwReqs* follow the idea of the derivative control, which sets its output according to the rate of change of the control error. We define and exemplify each type of *AwReq* in the following.

An ***aggregate AwReq*** refers to the instances of another requirement and imposes constraints on their success/failure rate. E.g., `AR2` is the simplest aggregate *AwReq*: it demands that the referred DA be true 99% of the time the goal *Receive emergency call* is attempted. Aggregate *AwReqs* can also specify the period of time to consider when aggregating requirement instances (e.g., `AR3`). The frequency with which the requirement is to be verified is an optional parameter for *AwReqs*. If it is omitted, then the designer is to select the frequency (if the period of time to consider has been specified, it can be used as default value for the verification frequency). `AR5` is an example of an *AwReq* with verification interval specified.

Another pattern for aggregate *AwReq* specifies the min/max success/failure a requirement is allowed to have (e.g., `AR4`). *AwReqs* can combine different requirements, like `AR5`, that integrates two QCs with different target rates. One can even compare the success counts of two requirements (`AR6`). This captures a desired property of the alternative selection procedure when deciding at runtime how to fulfill a goal.

`AR7` is an example of a ***trend AwReq*** that compare success rates over a number of periods. Trend *AwReqs* can be used to spot problems in how success/failure rates evolve over time. ***Delta AwReqs***, on the other hand, can be used to specify acceptable thresholds for the fulfillment of requirements, such as achievement time. `AR8` specifies that task *Update arrival at site* should be satisfied (successfully finish execution) within 10 minutes of completing task *Inform driver*. This means that once the dispatcher has informed the ambulance driver where the emergency is, she should arrive there within 10 minutes.

Another delta *AwReq*, `AR9`, shows how we can talk not only about success and failure of requirements, but about changes of states, following the state machine diagram of figure 3. In effect, when we say a requirement "should [not] succeed (fail)" we mean that it "should [not] transition from `Undecided` to `Succeeded` (`Failed`)". `AR9` illustrates yet another case: the task *Mark as unique or duplicate* should be decided — i.e., should leave the `Undecided` state — within 5 minutes. In other words, regardless if they succeeded or fail, operators should not spend more than 5 minutes deciding if a call is a duplicate of another call or not.

Finally, `AR10` and `AR11` are the examples of ***meta-AwReqs***: *AwReqs* that talk about other *AwReqs*. As we have previously discussed, *AwReqs* are based on the premise that even though we elicited, designed and implemented a system planning for all requirements to be satisfied, at runtime things might go wrong and requirements could fail, so *AwReqs* are added to trigger system adaptation in these cases. In this sense, *AwReqs* themselves are also requirements and, therefore, are also bound to fail at runtime. Thus, meta-*AwReqs* can provide further layers of adaptation in some cases if needed be.

One of the motivations for meta-*AwReqs* is the application of gradual reconciliation/compensations actions. This is the case with `AR10`: if `AR3` fails (i.e., *Search call database* has less than 95% success rate in a week), tagging the calls as "possibly ambiguous" (reconciling `AR3`) might be enough, but if `AR3`'s success rate considering the whole month is below 75% (e.g., it fails at least two out of four weeks), a deeper analysis of the database search problems might be in order (reconciling `AR10`). Another useful case for meta-*AwReqs* is to avoid executing specific reconciliation/compensation actions too many times. For example, `AR5` states that 60% of the ambulances should arrive in up to 10 minutes and 80% in up to 15 and to reconcile we should trigger messages to all users of the ADS. To avoid sending repeated messages in case it fails again, `AR11` states that `AR5` should never fail and, in case it does, its reconciliation decreases `AR5`'s percentages by 10 points (to 50% and 70%, respectively), which means that a new message will be sent only if the emergency response performance actually gets worse. If sending this message twice a month were to be avoided, `AR11`'s reconciliation could be, for example, disabling `AR5` for that month. As mentioned before, reconciliation is discussed in section 7.

With enough justification to do so, one could model an *AwReq* that refers to a meta-*AwReq*, which we would call a meta-meta-*AwReq* — or third-level *AwReq*. There is no limit on how many levels can be created, as long as meta-*AwReqs* from a given level refer strictly to *AwReqs* from lower levels, in order to avoid circular references. It is important to note that the name meta-*AwReq* is due only to the fact that it consists of an *AwReq* over another *AwReq*. This does not mean, however, that multiple levels of adaptation loops are required to monitor them. As will be presented in section 5, monitoring is operationalized by EEAT, which does so by matching method calls to invariants described in OCL$_{TM}$ (an example of this was presented in section 2.3), regardless of the class of the object that is receiving the message (goal, task, *AwReq*, meta-*AwReq*, etc.).

3.2 Patterns and Graphical Representation

Specifying *AwReqs* is not a trivial task. For this reason we propose *AwReq* patterns to facilitate their elicitation and analysis and a graphical representation that allows us to include them in the goal model, improving communication among system analysts and designers.

Many *AwReqs* have similar structure, such as "something must succeed so many times". By defining patterns for *AwReqs* we create a common vocabulary for analysts. Furthermore, patterns are used in the graphical representation of *AwReqs* in the goal model and code generation tools could be provided to automatically write the *AwReq* in the language of choice based on the pattern. In section 5.1, we provide OCL_{TM} idioms for this kind of code generation. We expect that the majority (if not all) *AwReqs* fall into these patterns, so their use can relieve requirements engineers from most of the specification effort.

Table 2 contains a list of patterns that we have identified so far in our research on this topic. This list is by no means exhaustive and each organization is free to define its own patterns (with their own names and meanings). We have already shown the pattern representation of the *AwReqs* that were elicited for the ADS in the last column of table 1. For such representation, we have used the patterns of table 2, mnemonics to refer to the requirements and abbreviated amounts of time

Table 2. A non-exhaustive list of *AwReq* patterns

Pattern	**Meaning**
`NeverFail(R)`	Requirement `R` should never fail. Analogous patterns `AlwaysSucceed`, `NeverCanceled`, etc.
`SuccessRate(R, r, t)`	`R` should have at least success rate `r` over time `t`.
`SuccessRateExecutions (R, r, n)`	`R` should have at least success rate `r` over the latest `n` executions.
`MaxFailure(R, x, t)`	`R` should fail at most `x` times over time `t`. Analogous patterns `MinFailure`, `MinSuccess` and `MaxSuccess`.
`ComparableSuccess(R, S, x, t)`	`R` should succeed at least `x` times more than `S` over time `t`.
`TrendDecrease(R, t, x)`	The success rate of `R` should not decrease `x` times consecutively considering periods of time specified by `t`. Analogous pattern `TrendIncrease`.
`ComparableDelta(R, S, p, x)`	The difference between the value of attribute `p` in requirements `R` and `S` should not be greater than `x`.
`StateDelta(R, s1, s2, t)`	`R` should transition from state `s1` to state `s2` in less time than what is specified in `t`.
P_1 `and` / `or` P_2; `not` P	Conjunction, disjunction and negation of patterns.

like in OCL$_{TM}$ timeouts [25]. Furthermore, it is important to note that when requirements engineer create patterns, they are responsible for their consistency and correctness and, unfortunately, our approach does not provide any tool to help in this task.

Given that *AwReqs* can be shortened by a pattern we propose they be represented graphically in the goal model along with other elements such as goals, tasks, softgoals, DAs and QCs. For that purpose, we introduce the notation shown in figure 4, which shows the goal model of the ADS with the addition of *AwReqs*, represented graphically in the model. *AwReqs* are represented by thick circles with arrows pointing to the element to which they refer and the *AwReq* pattern besides it. The first parameter of the pattern is omitted, as the *AwReq* is pointing to it. In case an *AwReq* does not fit a pattern, the analyst should write its name and document its specification elsewhere.

3.3 Sources of Awareness Requirements

Like other types of requirements, *AwReqs* must be systematically elicited. Since they refer to the success/failure of other requirements, their elicitation takes place after the basic requirements have been elicited and the goal model constructed. There are several common sources of *AwReqs* and, in this section, we discuss some of these sources. We do not, however, propose a systematic process for *AwReq* elicitation and requirements engineers should use existing requirement elicitation techniques to discover requirements that belong to this new class.

One obvious source consists of the goals that are critical for the system-to-be to fulfill its purpose. If the aim is to create a robust and resilient system, then there have to be goals/tasks in the model that are to be achieved/executed at a consistently high level of success. Such a subset of critical goals can be identified in the process and *AwReqs* specifying the precise achievement rates that are required for these goals will be attached to them. This process can be viewed as the operationalization of high-level non-functional requirements (NFRs) such as Robustness, Dependability, etc. For example, the task *Input emergency information* is critical for this process since all subsequent activities depend on it. Also, government regulations and rules may require that certain goals cannot fail or be achieved at high rates. Similarly, *AwReqs* are applied to DAs that are critical for the system (e.g., *Communications networks working*).

As shown in section 3.1, *AwReqs* can be derived from softgoals. There, we presented a QC *Ambulance arrives in 10 minutes* that metricizes a high-level softgoal *Fast assistance*. Then, *AwReq* `AR5` is attached to it requiring the success rate of 60%. This way the system is able to quantitatively evaluate at runtime whether the quality requirements are met over large numbers of process instances and make appropriate adjustments if they are not.

Qualitative softgoal contribution labels in goal models capture how goals and tasks affect NFRs, which is helpful, e.g., for the selection of the most appropriate alternatives. In the absence of contribution links, *AwReqs* can be used to capture the fact that particular goals are important or even critical to meet NFRs and thus those goals' high rate of achievement is needed. This can be viewed as

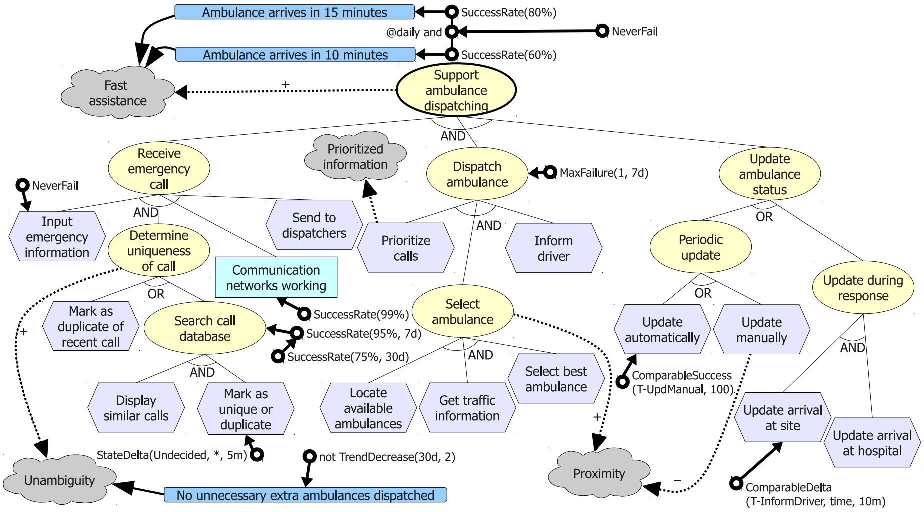


Fig. 4. Goal model of figure 1 with *AwReqs* represented graphically

an operationalization of a contribution link. For example, the task *Prioritize calls* in figure 1 positively affects the softgoal *Prioritized information* and can even be considered critical with respect to that softgoal. So, an *AwReq*, say, *SuccessRate(Prioritize Calls, 90%)*, can be added to the model to capture that fact. On the other hand, if a goal has a negative effect on an NFR, then an *AwReq* could demand a low success rate for it.

In Tropos [5] and other variations of goal modeling notation, alternatives introduced by OR-decomposed goals are frequently evaluated with respect to certain softgoals. The goal *Periodic updates* in figure 1 (of figure 4) is such an example. The evaluations are qualitative and show whether alternatives contribute positively or negatively to softgoals. In our approach, softgoals are refined into QCs and the qualitative contribution links are removed. However, the links do capture valuable information on the relative fitness of alternative ways to achieve goals. *AwReqs* can be used as a tool to make sure that "good" alternatives are still preferred over bad ones. For instance, the *AwReq* `AR6` states that automatic updates must be executed more often than manual ones, presumably because this is better for proximity of ambulances to target locations and due to the costs of manual updates. This way the intuition behind softgoal contribution links is preserved. If multiple conflicting softgoals play roles in the selection of alternatives, then a number of alternative *AwReqs* can be created since the selection of the best alternative will be different depending on the relative priorities of the conflicting NFRs.

One of the difficulties with *AwReqs* elicitation is coming up with precise specifications for the desired success rates over certain number of instances or during a certain time frame. To ease the elicitation and maintenance we recommend a gradual elicitation, first using high-level qualitative terms such as "medium" or “high” success rate, “large” or “medium” number of instances, etc. Thus, the *AwReq* may originate as “high success rate of `G` over medium number of instances” before becoming *SuccessRate(G, 95%, 500)*. Of course, the quantification of these high-level terms is dependent on the domain and on the particular *AwReq*. So, “high success rate” may be mapped to 80% in one case and to 99.99% in another. Additionally, using abstract qualitative terms in the model while providing the mapping separately helps with the maintenance of the models since the model remains intact while only the mapping is changing.

4 Specifying Awareness Requirements

We have just introduced *AwReqs* as requirements that refer to the success or failure of other requirements. This means that the language for expressing *AwReqs* has to treat requirements as first class citizens that can be referred to. Moreover, the language has to be able to talk about the status of particular requirements instances at different time points. We have chosen to use an existing language, namely OCL_{TM}, over creating a new one, therefore inheriting its syntax and semantics. The subset of OCL_{TM} features available to requirements engineers when specifying *AwReqs* is the subset supported by the monitoring framework, EEAT, introduced in section 2.3. A formal definition of the syntax and the semantics of *AwReqs* is out of the scope of this paper.

Our general approach to using it is as follows: (i) design-time requirements — as shown in figure 1, but also the *AwReqs* of table 1 — are represented as UML classes, (ii) run-time instances of requirements, such as various ambulance dispatch requests, are represented as instances of these classes. Representing system requirements (previously modeled as a goal model) in a UML class diagram is a necessary step for the specification of *AwReqs* in any OCL-based language, as OCL constraints refer to classes and their instances, attributes and methods. Even though other UML diagrams (such as the sequence diagram or the activity diagram) might seem like a better choice for the representation of requirements and *AwReqs*, having instances of classes that represent requirements at runtime is mandatory for the OCL-based infrastructure that we have chosen.

Hence, we present in figure 5 a model that represents classes that should be extended to specify requirements. In other words, each requirement of our system should be represented by a UML class, extending the appropriate class from the diagram of figure 5. These classes have the same name as the mnemonics used in the pattern column of table 1. Moreover, the first letter of each class name indicates which element of figure 5 is being extended (`T` for `Task`, `G` for `Goal` and so forth). Note that the diagram of figure 5 does not represent a *meta-model* for requirements due to the fact that the classes that represent the system requirements are subclasses of the classes in this diagram, not instances of them as it is the case with meta-models. This inheritance is necessary in order for *AwReq* specifications to be able to refer to the methods defined in these classes, as they are inherited by the requirement classes.

Another important observation is that these classes are only an abstract representation of the elements of the goal model (figure 1) and they are part of the

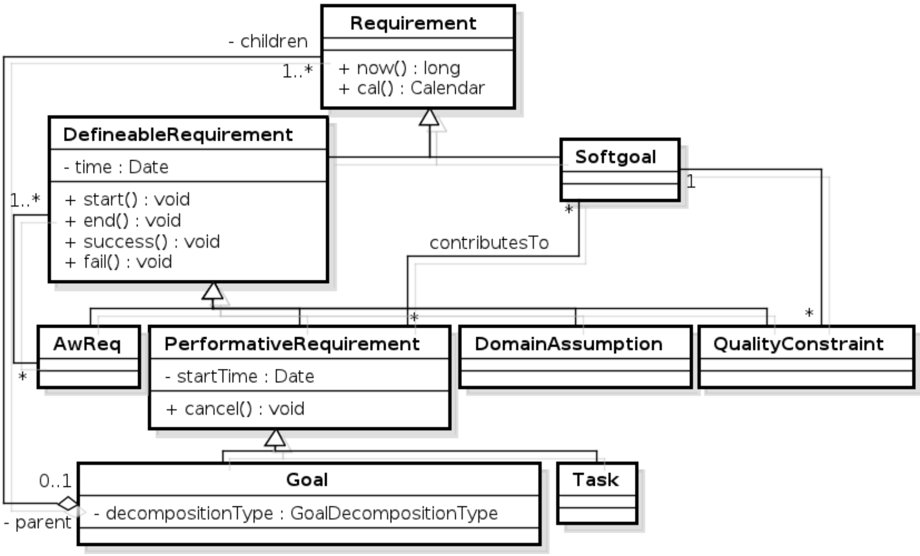


Fig. 5. Class model for requirements in GORE

monitoring framework that will be presented in section 5. They are not part of the monitored system (i.e., the ADS). In other words, the actual requirements of the system are not implemented by means of these classes.

Figure 6 shows the specification of some *AwReqs* of table 1 using OCL_{TM}. For example, consider `AR1`, which refers to a UML `Task` requirement. Figure 6 presents `AR1` as an OCL invariant on the class `T-InputInfo`, which should be a subclass of `Task` (from figure 5) and represents requirement *Input emergency information*. The invariant dictates that instances of `T-InputInfo` should never be in the `Failed` state, i.e., *Input emergency information* should never fail.

Aggregate *AwReqs* place constraints over a collection of instances. In `AR3`, for example, all instances of `G-SearchCallDB` executed in the past 7 days are retrieved in a set named `week` (using date comparison as in [25]), then we use the `select()` operation again to separate the subset of the instances that succeeded and, finally, we compare the sizes of these two sets in order to assert that 95% of the instances are successful at all times (`always`).

Trend *AwReqs* are similar, but a bit more complicated as we must separate the requirements instances into different time periods. For `AR7`, the `select()` operation was used to create sets with the instances of `Q-NoExtraAmb` for the past three months to compare the rate of success over time.

Delta *AwReqs* specify invariants over single instances of the requirements. `AR8` singles out the instances of `T-UpdAtSite` that are related to `T-InformDriver` in the `related` set by comparing the `callID` argument using OCL_{TM}'s `arguments()` operation [25]. Its invariant states that eventually the `related` set should have exactly one element, which should both be successful and finish its execution within 10 minutes of `T-InformDriver`'s end time.

```
context T-InputInfo inv AR1: never(self.oclInState(Failed))

context G-SearchCallDB
   def: week : G-SearchCallDB.allInstances()->select(g | new Date().diff(g.time, DAYS) <= 7)
   def: success : week->select(d | d.oclInState(Succeeded))
   inv AR3: always(success->size() / week->size() >= 0.95)

context Q-NoExtraAmb
   def: all : Set = Q-NoExtraAmb.allInstances()
   def: now : Date = new Date()
   def: m1 : Set = all->select(q | now.diff(q.time, DAYS) <= 30)
   def: m2 : Set = all->select(q | (now.diff(q.time, DAYS) <= 60) and (now.diff(q.time, DAYS > 30)))
   def: m3 : Set = all->select(q | (now.diff(q.time, DAYS) <= 90) and (now.diff(q.time, DAYS > 60)))
   def: success1 : Set = m1->select(q | q.oclInState(Succeeded))
   def: success2 : Set = m2->select(q | q.oclInState(Succeeded))
   def: success3 : Set = m3->select(q | q.oclInState(Succeeded))
   inv AR7: never(((success3->size() / m3->size()) < (success2->size() / m2->size())) and
      ((success2->size() / m2->size()) < (success1->size() / m1->size())))

context T-InformDriver
   def: related : Set = T-UpdAtSite.allInstances()->select(t | t.arguments('callID') = self.arguments('callID'))
   inv AR8: eventually(related->size() == 1) and always(related->forAll(t | t.oclInState(Succeeded) and
      t.time.diff(self.time, MINUTES) <= 10))

context T-MarkUnique
   inv AR9: eventually(not self.oclInState(Undecided)) and never(self.time.diff(self.startTime, MINUTES) > 5)
```

Fig. 6. Examples of *AwReqs* expressed in OCL_{TM}

AR9 shows how to specify the example in which we do not talk specifically about success or failure of a requirement, but its change of state: eventually tasks T-MarkUnique should not be in the Undecided state and the difference between their start and end times should be at most 5 minutes.

5 Implementation and Evaluation

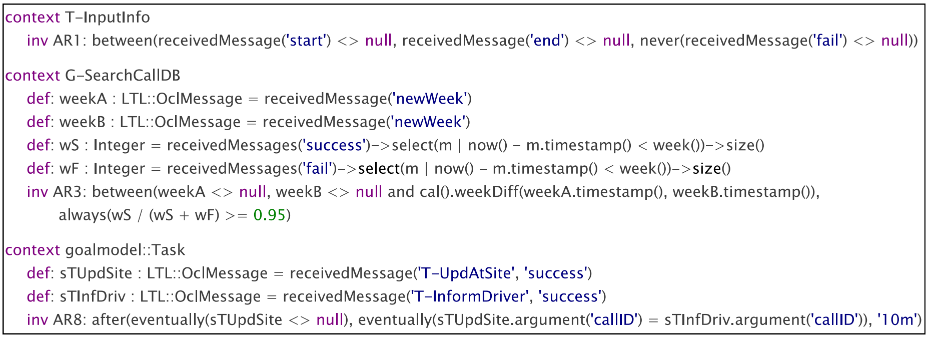
To evaluate our proposal we have implemented a framework to monitor *AwReqs* at runtime. Such evaluation considers three aspects of this framework:

1. Can *AwReqs* be monitored? Specifically, can an automated monitor evaluate requirements types enumerated in table 2 at runtime? Applying a constructive experiment, we show this is true (§5.1);
2. Can the *AwReqs* framework provide value for the analysis of a real system? With simulation experiments, we demonstrate this is true for scenarios of the ADS (§5.2);
3. What is the impact of *AwReqs* monitoring in the overall performance of the monitored system? We discuss this in §5.3.

The first two items above represent the experimental and descriptive evaluation methods of Design Science, as enumerated by [16]. After this initial evaluation, two other experiments were conducted, modeling the *AwReqs* of systems that are close to real-world applications: an Adaptive Computer-aided Ambulance Dispatch system [31] that is somewhat similar to the ADS, but was based on the requirements for the London Ambulance System Computer-Aided Despatch (LAS-CAD) [1]; and an Automatic Teller Machine [35]. Since these experiments involved simulations of running systems based on their requirements models, future evaluation efforts include experiment with actual running systems and conducting full-fledged case studies with partners in industry.

5.1 Monitoring Awareness Requirements Patterns

As mentioned in section 2.3, we have used EEAT to monitor *AwReqs* expressed in OCL$_{TM}$. In its current version, EEAT compiles the OCL$_{TM}$ expression into a rule file that is triggered by messages exchanged by objects at runtime (i.e., method calls). For this reason, we have to transform the initial specification of the *AwReqs* to one based on methods received by the run-time instances which represent the requirements. Figure 7 shows some of the *AwReqs* previously presented in figure 6 in their "EEAT specifications".



```
context T-InputInfo
   inv AR1: between(receivedMessage('start') <> null, receivedMessage('end') <> null, never(receivedMessage('fail') <> null))

context G-SearchCallDB
   def: weekA : LTL::OclMessage = receivedMessage('newWeek')
   def: weekB : LTL::OclMessage = receivedMessage('newWeek')
   def: wS : Integer = receivedMessages('success')->select(m | now() - m.timestamp() < week())->size()
   def: wF : Integer = receivedMessages('fail')->select(m | now() - m.timestamp() < week())->size()
   inv AR3: between(weekA <> null, weekB <> null and cal().weekDiff(weekA.timestamp(), weekB.timestamp()),
        always(wS / (wS + wF) >= 0.95)

context goalmodel::Task
   def: sTUpdSite : LTL::OclMessage = receivedMessage('T-UpdAtSite', 'success')
   def: sTInfDriv : LTL::OclMessage = receivedMessage('T-InformDriver', 'success')
   inv AR8: after(eventually(sTUpdSite <> null), eventually(sTUpdSite.argument('callID') = sTInfDriv.argument('callID')), '10m')
```

Fig. 7. Specification of *AwReqs* for EEAT

For monitoring to work, then, the source code of the monitored system (in this case, the ADS) has to be instrumented in order to create the instances of the classes that represent the requirements at runtime and call the methods defined in classes `DefinableRequirement` and `PerformativeRequirement` from figure 5. Methods `start()` and `end()` should be called when the system starts and ends the execution of a goal or task (or the evaluation of a QC or DA), respectively. Together with the `between` clause (one of Dwyer et al. scopes, see §2.3), these methods allow us to define the period in which *AwReqs* should be evaluated, because otherwise the rule system could wait indefinitely for a given message to arrive.

Given the right scope, the methods `success()`, `fail()` and `cancel()` are called by the monitored system to indicate a change of state in the requirement from `Undecided` to one of the corresponding final states (see figure 3). These methods are then used in the "EEAT specification" of *AwReqs*. For example, we define `AR1` not as never being in the `Failed` state, but as never receiving the `fail()` message in the scope of a single execution (between `start()` and `end()`).

An aggregate requirement, on the other hand, aggregates the calls during the period of time defined in the *AwReq*. For `AR3`, this is done by monitoring for calls of the `newWeek()` method, which are called automatically by the monitoring framework at the beginning of every week. Similar methods for different time periods, such as `newDay()`, `newHour()` and so forth, should also be implemented.

The last example shows the delta *AwReq* `AR8`, which uses OCL$_{TM}$ timeouts to specify that the `success()` method should be called in the `T-InformDriver` instance within 10 minutes after the same method is called in `T-UpdAtSite`, given that both instances refer to the same call ID, an argument that can be passed along the method. This can be implemented by having a collection of key-value pairs passed as parameters to the methods `start()`, `success()`, etc.

An automatic translator from the *AwReqs*' initial specification to their "EEAT specification" could be built to aid the designer in this task. Another possibility is to go directly from the *AwReq* patterns presented in section 3.2 to this final specification. Table 3 illustrates how some of the patterns of table 1 can be expressed in OCL$_{TM}$. These formulations are consistent with those shown in figure 7. The definitions and invariants are placed in the context of UML classes that represent requirements (see §4). For example, a `receiveMessage('fail')` for context `R`, denotes the called operation `R.fail()` for class `R`. Therefore, the invariant `pR` in the first row of table 3 is true if `R.fail()` is never called.

Table 3. EEAT/OCL$_{TM}$ idioms for some patterns

Pattern	**OCL$_{TM}$ idiom**

NeverFail(R)	`def: rm: OclMessage = receiveMessage('fail')`
	`inv pR: never(rm)`

SuccessRate(R, r, t)	`def: msgs: Sequence(OclMessage) = receiveMessages()->`
	`   select(range().includes(timestamp()))`
	`-- Note: these definitions are patterns that are assumed in`
	`    the following definitions`
	`def: succeed: Integer = msgs->select(methodName = 'succeed'))->size()`
	`def: fail: Integer = msgs->select(methodName = 'fail'))->size()`
	`inv pR: always(succeed / (succeed + fail) > r)`

ComparableSuccess (R, S, x, t)	`-- c1 and c2 are fully specified class names`
	`inv pR: always(c1.succeed > c2.succeed * x)`

MaxFailure(R, x, t)	`inv pR: always(fail < x)`

P_1 and/or P_2; not P	`-- arbitrary temporal and real-time logical expressions are`
	`    allowed over requirements definitions and run-time objects`

Of course, the patterns of table 1 represent only common kinds of expressions. *AwReqs* contain the range of expressions where a requirement `R1` can express properties about requirement `R2`, which include both design-time and run-time requirements properties. OCL$_{TM}$ explicitly supports such references, as the following expressions illustrate:

```
def: p1: PropertyEvent = receivedProperty('p:package.class.invariant')
inv p2: never(p1.satisfied() = false)
```

In OCL$_{TM}$, all property evaluations are asserted into the run-time evaluation repository as `PropertyEvent` objects. The definition expression of `p1` refers to an invariant (on a UML class, in a UML package). Properties about `p1` include its run-time evaluation (`satisfied()`), as well as its design-time properties (e.g., `p1.name()`). Therefore, in OCL$_{TM}$, requirements can refer to their design-time and run-time properties and, thus, *AwReqs* can be represented in OCL$_{TM}$.

To determine if the *AwReq* patterns can be evaluated at runtime, we constructed scenarios for each row of table 3. Each scenario includes three alternatives, which should evaluate to true, false, and indeterminate (non-false) during requirements evaluation. We had EEAT compile the patterns and construct a monitor. Then, we ran the scenarios. In all cases, EEAT correctly evaluated the requirements.

To illustrate how EEAT evaluates OCL$_{TM}$ requirements in general, the next subsection describes in detail a portion of the evaluation of the ADS' monitoring system, which was generated from the requirements of table 1.

5.2 Evaluating an Awareness Requirement Scenario

The requirements of the ADS provide a context to evaluate the *AwReq* framework. The ADS is implemented in Java. Its requirements (table 1) are represented as OCL$_{TM}$ properties, using patterns like those presented in table 3 and figure 7. Scenarios were developed to exercise each requirement so that each of them should evaluate as failed or succeeded. When each scenario is run, EEAT evaluates the requirements and returns the correct value. Thus, all the scenarios that test ADS requirements presented here evaluate correctly.

Next, we describe how this process works for one requirement and one test. Consider a single vertical slice of the development surrounding requirement `AR1`, as shown in figure 8:

1. Analysts specify the *Emergency input information* task of figure 1 (i.e., `T-InputInfo`) as a task specification (e.g., input, output, processing algorithm) along with *AwReqs* such as `AR1`;
2. Developers produce an input form and a processor fulfilling the specification. In a workflow system architecture, `T-InputInfo` is implemented as a XML form which is processed by a workflow engine. In our standard Java application, `T-InputInfo` is implemented as a form that is saved to a database. In any case, the point at which the input form is processed is the instrumentation point;
3. Validators (i.e., people performing requirements monitoring) instrument the software. Five events are logged in this simple example: (a) `T-InputInfo .start()`, (b) `T-InputInfo.end()`, (c) `T-InputInfo.success()`, (d) `T-InputInfo.fail()`, and (e) `T-InputInfo.cancel()`. Of course, the developers may have chosen a different name for `T-InputInfo` or the five methods, in which case, the validator must introduce a mapping from the run-time object and methods to the requirements classes and operations. Given the

rise of domain-driven software development, in which requirements classes are implemented directly in code, the mapping function is often relatively simple — even one-to-one;

4. The EEAT monitor continually receives the instrumented events and determines the satisfaction of requirements. In the case of `AR1`, if the `T-InputInfo` form is processed as succeed or cancel, then `AR1` is true.

The architecture and process of EEAT provides some context for the preceding description. EEAT follows a model-driven architecture (MDA). It relies on the Eclipse Modeling Framework (EMF) for its meta-model and the OSGi component specifications. This means that the OCL_{TM} language and parser is defined as a variant of the Eclipse OCL parser by providing EMF definitions for operations, such as `receivedMessage`. The compiler generates Drools rules, which combined with the EEAT API, provide the processing to incrementally evaluate OCL_{TM} properties at runtime.

EEAT provides an Eclipse-based UI. However, the run-time operates as a OSGi application, comprised as a dynamic set of OSGi components. For these experiments, the EEAT run-time components consist of the OCL_{TM} property evaluator, compiled into a Drools rule system, and the EEAT log4j feed, which listens for logging events and adds them to the EEAT repository. The Java application was instrumented by Eclipse TPTP to send CBE events via log4j to EEAT, where the event are evaluated by the compiled OCL_{TM} property monitors. For a more complete description of the language and process of EEAT, see [26,27].

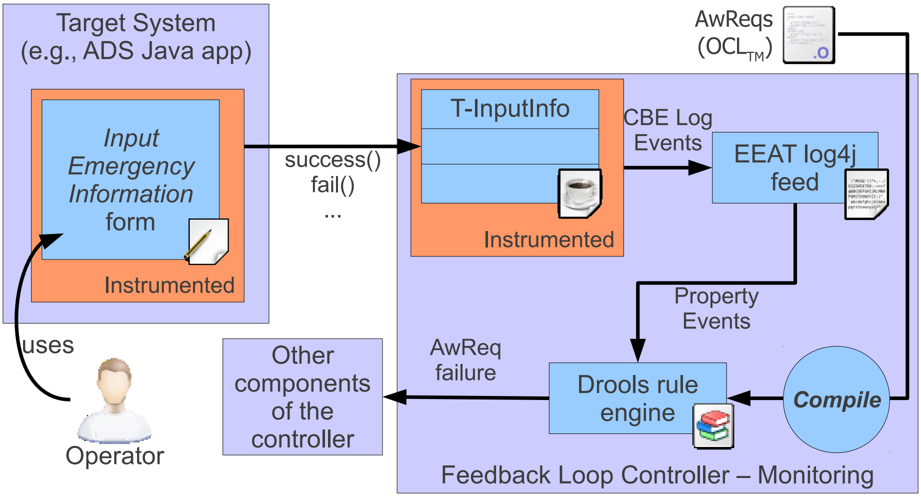


Fig. 8. Overview of the *AwReqs* monitoring framework

5.3 Monitor Performance

Monitoring has little impact on the target system, mostly because the target system and the monitor typically run on separate computers. The TPTP Probekit provides optimized byte-code instrumentation, which adds little overhead to some (selected) method calls in the target system. The logging of significant events consumes no more than 5%, and typically less than 1% overhead.

For real-time monitoring, it is important to determine if the target events can overwhelm the monitoring system. A performance analysis of EEAT was conducted by comparing the total monitoring runtime vs. without monitoring using 40 combinations of the Dwyer et al. temporal patterns [11]. For data, a simple two-event sequence was the basis of the test datum; for context, consider the events as an arriving email and its subsequent reply. These pairs were continuously sent to the server 10,000 times. In the experiment, the event generator and EEAT ran in the same multi-threaded process. The test ran as a JUnit test case within Eclipse on a Windows Server 2003 dual core 2.8 GHz with 1G memory. The results suggest that, within the test configuration, sequential properties (of length 2) are processed at 137 event-pairs per second [26]. This indicates that EEAT is reasonably efficient for many monitoring problems.

6 Related Work

In the literature, there are many approaches for the design of adaptive systems. A great deal of them, however, focus on architectural solutions for this problem, such as the Rainbow framework [13], the proposal of Kramer & Magee [19], the work of Sousa et al. [30], the SASSY framework [22], among others. These approaches usually express adaptation requirements in a quantitative manner (e.g., utility functions) and focus on quality of service (i.e., non-functional requirements). In comparison, our research is focused on early requirements (goal) models, allowing stakeholders and requirements engineers to reason about adaptation on a higher level of abstraction. Furthermore, *AwReqs* can be associated not only to non-functional characteristics of the system (represented by quality constraints), but also to functional requirements (goals, tasks) and even domain assumptions. The rest of this section focuses on recent approaches that share a common focus with ours in early requirements models.

A number of recent proposals offer alternative ways of expressing and reasoning about partial requirements satisfaction. RELAX by Whittle, et al. [36] is one such approach aimed at capturing uncertainty (mainly due to environmental factors) in the way requirements can be met. Unlike our goal-oriented approach, RELAX assumes that structured natural language requirements specifications (containing the SHALL statements that specify what the system ought to do) are available before their conversion to RELAX specifications. The modal operators available in RELAX, SHALL and MAY. . .OR, specify, respectively, that requirements must hold or that there exist requirements alternatives. We, on the other hand, capture alternative requirements refinement using OR decompositions of goals.

In RELAX, points of flexibility/uncertainty are specified declaratively, thus allowing designs based on rules, planning, etc. as well as to support unanticipated adaptations. Some requirements are deemed invariant — they need to be satisfied no matter what. This corresponds to the *NeverFail(R) AwReq* pattern in our approach. Other requirements are made more flexible in order to maintain their satisfaction by using "as possible"-type RELAX operators. Because of these, RELAX needs a logic with built-in uncertainty to capture its semantics. The authors chose fuzzy branching temporal logic for this purpose. It is based on the idea of fuzzy sets, which allows gradual membership functions. E.g., the function for fuzzy number 2 peaks at 1 given the value 2 and slopes sharply towards 0 as we move away from 2, thus capturing "approximately 2". Temporal operators such as *Eventually* and *Until* allow for temporal component in requirements specifications in RELAX.

Our approach is much simpler compared to RELAX. The *AwReqs* constructs that we provide just reference other requirements. Thus, we believe that it is more suitable, e.g., for requirements elicitation activities. Our specifications do not rely on fuzzy logic and do not require a complete requirements specification to be available prior to the introduction of *AwReqs*. Also, our language does not require complex temporal constructs. However, the underlying formalism used for *AwReqs* — OCL$_{TM}$ — provides temporal operators, as does EEAT, so temporal properties can be expressed and monitored. Most of the work on generating OCL$_{TM}$ specifications can be automated through the use of patterns.

With each relaxation RELAX associates "uncertainty factors": properties of the environment that can or cannot be monitored, but which affect uncertainty in achieving requirements. Our future work includes such integration of domain models in our approach.

Using *AwReqs* we can express approximations of many of the RELAX-ed requirements. For instance, `AR5` from table 1 can be used as a rough approximation of the requirement "ambulances must arrive at the scene AS CLOSE AS POSSIBLE to 10 minutes' time". The general pattern for approximating fuzzy requirements is to first identify a number of requirements that differ in their strictness, depending on our interpretation of what "approximately" means. E.g., `R1` = "ambulance arrives in 10 min", `R2` = "ambulance arrives in 12 min", `R3` = "ambulance arrives in 15 min". Then, we assign desired satisfaction levels to these requirements. For instance, we can set success rate for `R1` to 60% (as in `AR5`), `R2` to 80%, and `R3` to 100%. This means that all ambulances will have to arrive within 10–15 min from the emergency call. The *AwReq* will then look like `AR12` = *SuccessRate(R1, 60%) AND SuccessRate(R2, 80%) AND SuccessRate(R3, 100%)*. On the other hand, `AR13` = *SuccessRate(R1, 80%) AND SuccessRate(R2, 100%)* provides a much stricter interpretation of the fuzzy duration with all ambulances required to arrive within 12 minutes.

Another related approach called FLAGS is presented in [3]. FLAGS requirements models are based on the KAOS framework [20] and are targeted at adaptive systems. It proposes crisp (Boolean) goals (specified in linear-time temporal logic, as in KAOS), whose satisfaction can be easily evaluated, and fuzzy goals that are specified using fuzzy constraints. In FLAGS, fuzzy goals are mostly associated with non-functional requirements. The key difference between crisp and fuzzy goals is that the former are firm requirements, while the latter are more flexible. Compared to RELAX, FLAGS is a goal-oriented approach and thus is closer in spirit to our proposal.

To provide semantics for fuzzy goals, FLAGS includes fuzzy relational and temporal operators. These allow expressing requirements such as something be almost always less than `X`, equal to `X`, within around `t` instants of time, lasts hopefully `t` instants, etc. As was the case with the RELAX approach, *AwReqs* can approximate some of the fuzzy goals of FLAGS while remaining quite simple. The example that we presented while discussing RELAX also applies here. Whenever a fuzzy membership function is introduced in FLAGS, its shape must be defined by considering the preferences of stakeholders. This specifies exactly what values are considered to be "around" the desired value. As we have shown above with `AR12` and `AR13`, *AwReqs* can approximate this "tuning" of fuzzy functions while not needing fuzzy logic and thus remaining more accessible to stakeholders.

Additionally, in FLAGS, adaptive goals define countermeasures to be executed when goals are not attained, using event-condition-action rules. Using a similar approach, we have recently published a proposal to complement *AwReqs* with adaptation strategies that provide compensation for failures [33]. Discussion in section 3 illustrates how *AwReqs* and meta-*AwReqs* could be used to enact the required compensation behavior, including relaxation of desired success rates. We further comment on these aspects on section 7.2.

Letier and van Lamsweerde [21] present an approach that allows for specifying partial degrees of goal satisfaction for quantifying the impact of alternative designs on high-level system goals. Their partial degree of satisfaction can be the result of, e.g., failures, limited resources, etc. Unlike FLAGS and RELAX, here, a partial goal satisfaction is measured not in terms of its proximity to being fully satisfied, but in terms of the probability that it is satisfied. The approach augments KAOS with a probabilistic layer. Here, goal behavior specification (in the usual KAOS temporal logic way) is separate from the quantitative aspects of goal satisfaction (specified by quality variables and objective functions). Objective functions can be quite similar to *AwReqs*, except they use probabilities. For instance, one such function presented in [21] states that the probability of ambulance response time of less than 8 min should be 95%. Objective functions are formally specified using a probabilistic extension of temporal logic. An approach for propagating partial degrees of satisfaction through the model is also part of the method.

Overall, the method can be used to estimate the level of satisfaction of high-level goals given statistical data about the current or similar system (from rather low-level measurable parameters). Our approach, on the other hand, naturally leads to high-level monitoring capabilities that can determine satisfaction levels for *AwReqs*.

There is a fundamental difference between the approaches described above and our proposal. There, by default, goals are treated as invariants that must always be achieved. Non-critical goals — those that can be violated from time to time — are relaxed. Then, the aim of those methods is to provide the machinery to conclude at runtime that while the system may have failed to fully achieve its relaxed goals, this is acceptable. So, while relaxed goals are monitored at runtime, invariant ones are analyzed at design time and must be guaranteed to always be achievable at runtime.

In our approach, on the other hand, we accept the fact that a system may fail in achieving any of its initial (stratum 0) requirements. We then suggest that critical requirements are supplemented by *AwReqs* that ultimately lead to the introduction of feedback loop functionality into the system to control the degree of violation of critical requirements. Thus, the feedback infrastructure is there to reinforce critical requirements and not to monitor the satisfaction of expendable (i.e., relaxed) goals, as in RELAX/FLAGS. The introduction of feedback loops in our approach is ultimately justified by criticality concerns.

7 From Awareness Requirements to Feedback Loops

As stated in section 1, our intention in this proposal is to identify and explore requirements that lead to the introduction of feedback loop functionality into adaptive systems. In section 3.3, we discussed the sources of *AwReqs*, while section 5 explained how EEAT can be used to monitor *AwReqs* at runtime to determine if they are attained or not. In this section, we present the overview of the role of Awareness Requirements in our overall approach for feedback loop-based requirements-driven adaptive systems design.

Figure 9 shows a variant of a feedback controller diagram adapted for requirements-driven adaptive systems. Here, system requirements play the role of the reference input, while indications of requirements convergence signaling if the

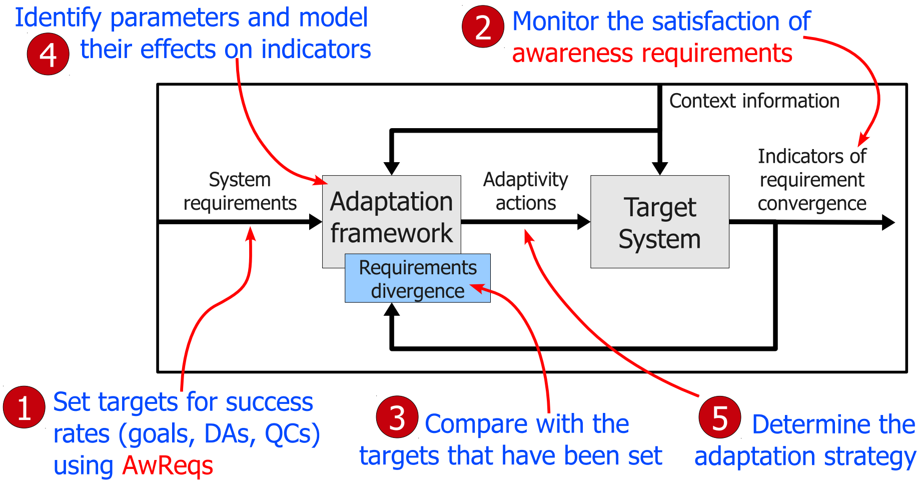


Fig. 9. A feedback loop illustrating the steps of the proposed process

requirements have been met replace the traditional monitored output of the controller. The controller itself is represented by a requirements-driven adaptation framework that controls the target system through executing adaptation actions that correspond to the control input in traditional feedback control schemes. Dynamically changing context corresponds to the disturbance input of the control loop. Finally, the measure of requirements divergence is the control error.

Furthermore, the phases of our proposed approach are added to the feedback loop diagram in figure 9, labeled 1 through 5. Step 1 is to set the targets for system to achieve/maintain at runtime. *AwReqs*, as discussed here, are used for this purpose. For step 2, the EEAT monitoring framework presented in section 5 is used to monitor whether the *AwReqs* are attained at runtime. Given the values for the *AwReq* attainment at runtime, in step 3 we calculate requirements divergence. If the targets are not met, this warrants a system adaptation. The system identification process (step 4) is aimed at linking system configuration parameters with indicators of requirements convergence and can be used to determine possible system reconfigurations. This process is further discussed in section 7.1. Finally, adaptation strategies/actions (step 5 in figure 9) are used by the adaptation framework to actually adapt the target system. These are further discussed in section 7.2.

7.1 System Identification

As we have shown throughout sections 3 to 5, *AwReqs* can be used to determine when requirements are not being satisfied, much the same way a control system calculates the control error, i.e., the discrepancy between the reference input (desired) and the measured output (outcome). The next step, then, is to determine the control input based on this discrepancy, i.e., determine what could be done to adapt the target system to ultimately satisfy the requirements.

In Control Theory (e.g., [15]), the first step towards accomplishing this is an activity called *System Identification*, which is the process of determining the equations that govern the dynamic behavior of a system. This activity is concerned with: (a) the identification of system parameters that, when manipulated, have an effect on the measured output; and (b) the understanding of the nature of this effect. Afterwards, these equations can guide the choice of the best way to adapt to different circumstances. For example, in a control system in which the room temperature is the measured output, turning on the air conditioner lowers the temperature, whereas using the furnace raises it. If the heating/cooling systems offer different levels of power, there is also a relation between such power level and the rate in which the temperature in the room changes.

In [32] we propose a systematic process for conducting *System Identification* for adaptive software systems, along with a language that can be used to represent how changes in system parameters affect the indicators of requirements convergence. After *AwReqs* have been elicited as *indicators*, the *System Identification* process consists of three activities:

1. **Identify parameters:** determine points of variability in the system (OR-decompositions, parameters related to system goals or tasks) whose change

of value affects any of the indicators. For instance, the set of required fields (an enumerated parameter) affects *AwReq* `AR1` (see table 1) — less required fields makes inputting information easier; the number of ambulances, as well as operators and dispatchers working, affects *AwReq* `AR5` — the higher the number, the higher the chances of fast assistance;

2. **Identify relations:** for each indicator–parameter pair (not only the ones identified in the previous step, but the full $\{indicators\} \times \{parameters\}$ Cartesian product), verify if there is a relation between changes in the parameter and the value of the indicator. For each existing relation, model qualitative information about the nature of the effect using differential equations. For example, $\Delta\,(AR1/RequiredFields) < 0$ indicates that decreasing the required fields (assuming the enumerated values form a totally ordered set) increases the success of `AR1`; $\Delta\,(AR5/NumberOfAmbulances) > 0$ states that increasing the number of ambulances also increases the success of `AR5`;
3. **Refine relations:** after identifying initial relations, the model can be refined by comparing and combining those that refer to the same indicator. For example, $\Delta\,(AR5/NumberOfAmbulances) > \Delta\,(AR5/NumberOfOperators)$ tells us that buying more ambulances is more effective than hiring more operators when considering how fast ambulances get to emergency sites.

A more detailed explanation of the *System Identification* process and the proposed language for modeling relations between indicators and parameters can be found in [32]. However, the basic examples above already give us the intuition that this kind of information is very important in order to determine the best way to adapt the target system and, therefore, the models produced by *System Identification* can be used by the adaptation framework for this purpose. Adaptation strategies are discussed next.

7.2 Adaptation Strategies

There are several ways a system can be changed as a result of its failure to attain the requirements. We call one such possibility adaptation. Here, the system's configuration (the values of its parameters) is changed in attempt to achieve the indicator targets. This can be viewed as parameter tuning. There can be a number of possible reconfiguration strategies based on the amount of information available in the system identification model. The more information is available and the more quantitative it is, the more precise and advanced the reconfiguration strategies can become. The reconfigurations involve changing the values of the system parameter(s), which affect indicator(s) that failed to achieve their target values. With the absence of a fully quantitative model relating parameters and indicators, an adaptation strategy may involve a number of such reconfigurations that are performed in succession in attempt to bring the indictor value to its target. When more precise information is available, quantitative approaches, e.g., mimicking the PID controller [15] can be used. Detailed specification and analysis of these strategies is one of the subjects of our current research.

In addition to reconfiguring a system, *Evolution Requirements*, which describe evolutions of other requirements, can be used to identify specific changes to the system requirements under particular conditions (usually requirements failures, negative trends on achieving requirements, or opportunities for improvement). Unlike reconfigurations discussed above, evolution requirements may change the space of alternatives available for the system. In our recent work [33], we have identified a number of adaptation strategies, including *abort*, *retry*, *delegate* to an external agent, *relax/strengthen* the requirement, etc., constructed from the basic requirements evolution operations such as *initiate* (a requirement instance), *rollback* (changes due to an attempt to achieve a requirement), etc. These adaptation strategies can be applied at the requirements instance level (thus, fixing/improving a particular system instance) and/or type level (thereby changing the behavior of all subsequent system instances). Reconfiguration is considered as one possible adaptation strategy. It can be applied at both levels. Further, [33] proposes an ECA-based process for executing adaptation strategies in response to failures. Triggered by *AwReq* failures, this process attempts to execute the possibly many adaptation strategies associated with the *AwReq* in their preference order, while defaulting to the abort strategy if others do not prove successful.

We stress here that Awareness Requirements are absolutely crucial in our vision for requirements-driven adaptive systems design. They serve both as the means to specify targets to be met by the system (i.e., reference inputs for the feedback controller) and as the indicators of requirements convergence (i.e., the monitored outputs), with their failures triggering the above-described adaptation strategies.

8 Conclusions

The main contribution of this paper is the definition of a new class of requirements that impose constraints on the run-time success rate of other requirements. The technical details of the contribution include linguistic constructs for expressing such requirements (reference to other requirements, requirement states, temporal operators), expression of such requirements in OCL_{TM}, as well as portions of a prototype implementation founded on an existing requirements monitoring framework. We have also discussed the role of *AwReqs* in a complete process for the development of adaptive systems using a feedback loop-based adaptation framework that builds on top of this monitoring framework.

Other than working towards the full feedback loop implementation discussed in section 7, future steps in our research include the integration of domain models in the approach (as mentioned in section 6) and improvements in the definition and specification of *AwReqs*. Other questions also present themselves as opportunities for future work in the context of this research: what is the role of contextual information in this approach? How could we add predictive capabilities or probabilistic reasoning in order to avoid failures instead of adapting to them? Could this approach help achieve requirements evolution? These and other questions show how much work there is still to be done in this research area.

References

1. Report of the inquiry into the London Ambulance Service. South West Thames Regional Health Authority (1993)

2. Object Constraint Language, OMG Available Specification, Version 2.0 (2006), http://www.omg.org/cgi-bin/doc?formal/2006-05-01
3. Baresi, L., Pasquale, L., Spoletini, P.: Fuzzy Goals for Requirements-driven Adaptation. In: Proc. of the 18th IEEE International Requirements Engineering Conference, pp. 125–134. IEEE (2010)
4. Berry, D.M., Cheng, B.H.C., Zhang, J.: The Four Levels of Requirements Engineering for and in Dynamic Adaptive Systems. In: Proc. of the 11th International Workshop on Requirements Engineering: Foundation for Software Quality, pp. 95–100 (2005)
5. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An Agent-Oriented Software Development Methodology. Autonomous Agents and Multi-Agent Systems 8(3), 203–236 (2004)
6. 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.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009)
7. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
8. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed Requirements Acquisition. Science of Computer Programming 20(1-2), 3–50 (1993)
9. Dobson, S., et al.: A Survey of Autonomic Communications. ACM Transactions on Autonomous and Adaptive Systems 1(2), 223–259 (2006)
10. Doyle, J.C., Francis, B.A., Tannenbaum, A.R.: Feedback Control Theory. Macmillan Coll Div (1992)
11. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: Proc. of the 21st International Conference on Software Engineering, pp. 411–420. ACM (1999)
12. Flake, S.: Enhancing the Message Concept of the Object Constraint Language. In: Proc. of the 16th International Conference on Software Engineering & Knowledge Engineering, pp. 161–166 (2004)
13. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. Computer 37(10), 46–54 (2004)
14. Giese, H., Cheng, B.H.C. (eds.): Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. ACM (2011)
15. Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D.M.: Feedback Control of Computing Systems, 1st edn. Wiley (2004)
16. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design Science in Information Systems Research. MIS Quarterly 28(1), 75–105 (2004)
17. Jureta, I., Mylopoulos, J., Faulkner, S.: Revisiting the Core Ontology and Problem in Requirements Engineering. In: Proc. of the 16th IEEE International Requirements Engineering Conference, pp. 71–80. IEEE (2008)
18. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
19. Kramer, J., Magee, J.: A Rigorous Architectural Approach to Adaptive Software Engineering. Journal of Computer Science and Technology 24(2), 183–188 (2009)

20. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications, 1st edn. Wiley (2009)
21. Letier, E., van Lamsweerde, A.: Reasoning about Partial Goal Satisfaction for Requirements and Design Engineering. In: Proc. of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, vol. 29, pp. 53–62. ACM (2004)
22. Menasce, D.A., Gomaa, H., Malek, S., Sousa, J.A.P.: SASSY: A Framework for Self-Architecting Service-Oriented Systems. IEEE Software 28(6), 78–85 (2011)
23. Parashar, M., Figueiredo, R., Kiciman, E.E. (eds.): Proceedings of the 7th International Conference on Autonomic Computing. ACM (2010)
24. Robinson, W.N.: A requirements monitoring framework for enterprise systems. Requirements Engineering 11(1), 17–41 (2006)
25. Robinson, W.N.: Extended OCL for Goal Monitoring. Electronic Communications of the EASST 9 (2008)
26. Robinson, W.N., Fickas, S.: Designs Can Talk: A Case of Feedback for Design Evolution in Assistive Technology. In: Lyytinen, K., Loucopoulos, P., Mylopoulos, J., Robinson, B. (eds.) Design Requirements Engineering. LNBIP, vol. 14, pp. 215–237. Springer, Heidelberg (2009)
27. Robinson, W.N., Purao, S.: Monitoring Service Systems from a Language-Action Perspective. IEEE Transactions on Services Computing 4(1), 17–30 (2011)
28. Rohleder, C., Smith, J., Dix, J.: Requirements Specification - Ambulance Dispatch System. Tech. rep., Software Engineering (CS 3354) Course Project, University of Texas at Dallas, USA (2006), `http://www.utdallas.edu/~cjr041000/`
29. Rosenthal, D.: Consciousness and Mind, 1st edn. Oxford University Press (2005)
30. Sousa, J.P., Balan, R.K., Poladian, V., Garlan, D., Satyanarayanan, M.: A Software Infrastructure for User–Guided Quality–of–Service Tradeoffs. In: Cordeiro, J., Shishkov, B., Ranchordas, A., Helfert, M. (eds.) ICSOFT 2008. CCIS, vol. 47, pp. 48–61. Springer, Heidelberg (2009)
31. Souza, V.E.S.: An Experiment on the Development of an Adaptive System based on the LAS-CAD. Tech. rep., University of Trento (2012), `http://disi.unitn.it/~vitorsouza/a-cad/`
32. Souza, V.E.S., Lapouchnian, A., Mylopoulos, J.: System Identification for Adaptive Software Systems: A Requirements Engineering Perspective. In: Jeusfeld, M., Delcambre, L., Ling, T.-W. (eds.) ER 2011. LNCS, vol. 6998, pp. 346–361. Springer, Heidelberg (2011)
33. Souza, V.E.S., Lapouchnian, A., Mylopoulos, J.: (Requirement) Evolution Requirements for Adaptive Systems. In: Proc. of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 155–164. IEEE (2012)
34. Souza, V.E.S., Mylopoulos, J.: From Awareness Requirements to Adaptive Systems: a Control-Theoretic Approach. In: Proc. of the 2nd International Workshop on Requirements@Run.Time, pp. 9–15. IEEE (2011)
35. Tallabaci, G.: System Identification for the ATM System. Master thesis, University of Trento (to be submitted, 2012)
36. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.-M.: RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In: Proc. of the 17th IEEE International Requirements Engineering Conference, pp. 79–88. IEEE (2009)
37. Yu, E.S.K., Giorgini, P., Maiden, N., Mylopoulos, J.: Social Modeling for Requirements Engineering, 1st edn. MIT Press (2011)

Self-management of Distributed Systems Using High-Level Goal Policies

Liliana Rosa[1], Luís Rodrigues[1], and Antónia Lopes[2]

[1] INESC-ID, Instituto Superior Técnico, Universidade Técnica de Lisboa
lrosa@gsd.inesc-id.pt, ler@ist.utl.pt
[2] LASIGE, Faculty of Sciences, University of Lisbon
mal@di.fc.ul.pt

Abstract. A growing number of systems have to face dynamic and unpredictable execution conditions and workloads. In an attempt to address the challenges posed by such environments, many systems are built from customizable components. These components can be tuned according to the execution context, allowing the system to meet its QoS requirements. In this context, self-management is an essential quality. This chapter presents an approach for the self-management of systems built from customizable components based on high-level goal policies. With this approach, in response to changes in the execution context, the necessary system adaptations are automatically selected and deployed. The evaluation of different aspects of the approach relies on a web-based application deployed as a distributed clustered-based architecture.

1 Introduction

Today, most software systems must be designed and built to offer good performance on a wide range of operational envelopes, characterized by different settings such as a wide diversity of user profiles, dynamic workloads, or variable network conditions. To tackle different operational envelopes, these systems are often built from customizable components. Each component offers a range of configurable aspects that can be tuned to implement different tradeoffs between the service provided and the resources consumed. In this manner, the global system performance can be optimized such that it best matches the quality of service (QoS) requirements defined for a given target deployment.

In the majority of cases, the customizable options are determined offline according to an average expected usage pattern. Unfortunately, more and more often, the execution scenario is dynamic and unpredictable, which renders the pre-configuration inappropriate and results in poor performance and ineffective resource consumption. For instance, large-scale networked applications are often subject to highly variable and unpredictable workloads, that make static resource allocation ineffective. Also, many distributed applications, such as multimedia applications, even with few participants and stable workloads, may be required to reconfigure during runtime to maintain a given level of quality of service in face of changing network conditions.

R. de Lemos et al. (Eds.): Self-Adaptive Systems, LNCS 7475, pp. 162–190, 2013.

One way of addressing the challenges of dynamic execution environments consists in building self-adaptive solutions. In these solutions, the execution conditions are continuously monitored and, when changes are observed, the customizable options of the system components are adapted during runtime (e.g., [5,10,3]). However, even if the principle underlying this strategy is quite simple, finding an appropriate adaptation strategy can be a surprisingly complex task. The difficulty stems from the combinatorial nature of matching multiple adaptations, in multiple components: there are many tradeoffs involved and an adaptation may cancel or amplify the effect of another adaptation. As a result, to manually balance all the tradeoffs in order to maintain or return the system to a desired performance behavior becomes a hard, complex, and error-prone task.

In this chapter, we describe an approach that automates the selection of the adaptations that should be performed in response to changes in the execution environment. To guide the selection process, the system manager has to define a policy that is specified in terms of high-level goals, which capture the expected behavior of the system in terms of performance and quality metrics. Therefore, in our approach, only the desired behavior of the system needs to be specified and the required adaptations are selected automatically by the system.

The work described in this chapter is an extension of the work presented in [16,17] to address distributed systems. Previous work only considered non-distributed systems or systems where a distributed component was encapsulated by an abstract non-distributed component. The augmentations proposed in this chapter target distributed systems where components can be local or distributed, and deployed in one or more nodes. Furthermore, we discuss how aspects such as the number of component instances, instance location, local and global effects of adaptations are modeled and accounted for in the selection process.

The contributions of this work are two-fold. One is the identification of the key aspects to describe goals, specify adaptations, and select adaptations when addressing distributed components. The other contribution is to identify how to perform the system monitoring and execution of the adaptations in a distributed setting.

The rest of the chapter is structured as follows. Section 2 introduces an example application that we use throughout the chapter for illustrating the key concepts followed by an overview of our approach in Section 3. Then, Section 4 details the underlying adaptation model whereas Section 5 describes how planning is carried out, based on high-level goals. In Section 6 we revisit the running example and report on evaluation results. Section 7 addresses the related work and we conclude this chapter with Section 8, by pointing out other aspects that need further investigation.

2 Example: High Traffic Web Cluster

To better illustrate the proposed approach, we begin by introducing the example application that will be used for the remainder of the chapter. The aplication is a website that is subject to dynamic and unpredictable load, often facing high traffic. The website is an online shopping store, that allows users to register for an account, browse the products catalog, and perform online orders. The users can either be private or business clients, with distinct contents being served.

The website is deployed in a clustered-based architecture, i.e., a web cluster, as shown in Figure 1. The clients make requests to a virtual IP served by a front-end that acts as a load balancer, distributing the requests among the available servers. Each server node of the cluster has its own web server. All cluster nodes may receive and process any request from any client (server nodes are not specialized). However, to improve the performance by exploring data locality, the load balancer may attempt to forward similar requests and requests from the same session to the same server.

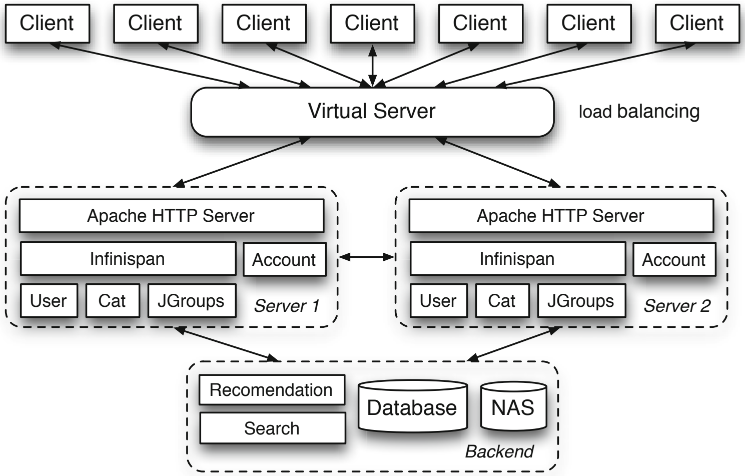


Fig. 1. The example application

Furthermore, a distributed in-memory caching system is collectively maintained by all servers. The cache is used to store results from recent requests. Therefore, when a server receives a request, it first checks if the request can be served from the cache. If this is not the case, then it forwards the request to the local software component that is capable of retrieving or generating the webpage. The cache is never used to store security-sensitive information, therefore some requests are forwarded directly to the corresponding software component.

Finally, all nodes have access to a shared persistent store, implemented by a database and additional network attached storage (NAS). These are part of the *backend*.

The website content is served by software components installed in the web servers deployed at each cluster node. The *Catalog* component (abbreviated by *Cat* in the figure) handles static content, such as product webpages. The

Account component handles sensitive content, such as credit card information or the user's account password. Finally, the *User* component handles dynamically generated content, which is customized to the user, such as product recommendations and customized searches. For that purpose it relies on two other centralized components. The *Recommendation* engine generates recommendations for a particular user and the *Search* engine gathers the results of a search customized to the user. These two components operate in the backend and can be accessed by any server. Each of the Catalog, Account, and User components are separated in *business* and *private* to cater to the different types of users, thus, totaling six main software components. From now on, when we refer to *Catalog*, *User*, or *Account* components, we are speaking of both *business* and *private* components.

The system runs in several machines. One of the machines is dedicated to the autonomic controller that self-manages the system (which is addressed in detail in Section 3). Each of the remaining machines executes the web server, an instance of the in-memory caching system, the multicast communication systems required by the cache, and the *Catalog*, *User* and *Account* components. Each server runs Linux OS and Apache HTTP server [1] and uses the RedHat Infinispan [11] as the distributed in-memory caching tool and JGroups [12] as the multicast service.

This is a very rich case study that offers many opportunities for self-adaptation. The first, and most obvious, is to adapt the number of active servers elastically in responses to changes in the workload. In periods of more load more servers can be activated and in periods of less load some servers can be assigned to other tasks or switched off for power saving.

However, the maximum number of servers is limited. Therefore, in periods of peak load it may be impossible to add more servers to the cluster of active nodes. In that case, one may rely on alternative configurations of the software components that serve requests at each node, avoiding overload at the servers. In fact, savings can be achieved at the expense of the quality of information provided to the clients. This can be achieved, for instance, by reducing the resolution of any media content that the pages may contain, or the freshness of the recommendations and search results presented to the user (by fetching the last generated recommendations/search results instead of generating a fresh one).

Finally, even in face of a constant rate of requests, the performance of the in-memory caching system is highly dependent on the profile of the workload, in particular, on the existence of hot-spots that may generate conflicts when accessing the cache entries. Thus, the cache performance may be optimized by selecting the most appropriate multicast primitive for each workload (basically, through the activation or inactivation of a mechanism able to totally order concurrent requests to the cache, preventing deadlocks at the expense of higher network utilization). This is an interesting aspect of the adaptation, as it illustrates several issues that are raised when performing dynamic adaptation on distributed components.

3 Approach Overview

The aim of the work proposed in this chapter is the self-management of software systems built from adaptable components, both distributed and non-distributed. The application described in the previous section is an example of a system that includes both types of components. In this section we present an overview of the proposed approach and discuss the major challenges that need to be addressed to implement goal-oriented self-adaptation of systems with distributed components. The solutions to these challenges will be addressed in detail in the subsequent sections of the chapter.

In our approach, similarly to other works, the self-management of a system is achieved using feedback. The system is augmented with suitable sensors and effectors that are connected to an external controller to implement a closed control loop [13,4,7]. The main activities performed by the control layer are (i) the collection of relevant data from sensors, (ii) the analysis of the collected data, (iii) the decision on how to adapt the system to reach a desirable state, and (iv) the implementation of the decision via the available effectors.

Central to our approach is the use of key performance indicators (KPIs), which are metrics that capture particular aspects of system performance and allow us to describe the system behavior. Any measurable aspect of the system operation can be captured by a KPI and the behavior of the control layer can be described in terms of the selected KPIs.

The control layer monitors the KPIs and adapts the managed system whenever the KPIs deviate from the desired values. KPIs are monitored using *sensors*, that send the information to a central element, the *monitor*. The monitor detects deviations in the system behavior and notifies the element that is responsible for deciding on how to adapt the system, the *planner*. When the planner decides an adaptation, the *executor* element performs the adaptation relying in several *effectors*, developed for the managed system. This architecture is depicted in Figure 2.

3.1 Monitor

The *monitor* collects data regarding some key aspects of the system behavior, i.e., the data required to compute the current value of the relevant system KPIs. In the example application, data such as the number of requests received per unit of time (request load) or requests served per unit of time (achieved throughput) are examples of relevant sensed data required to compute KPIs that capture the performance of the system. In terms of resource consumption, data such as consumed power, CPU, and memory can be used to define the KPIs. Still considering the example application, it is also possible to define KPIs that capture the quality of the service being provided to the end user, for instance, by using data regarding the quality of the images being produced, or the average time to serve a request.

The data collected by the monitor is captured by *sensors* in the managed system. Different sensors may have different scopes, as depicted in Figure 2. Some

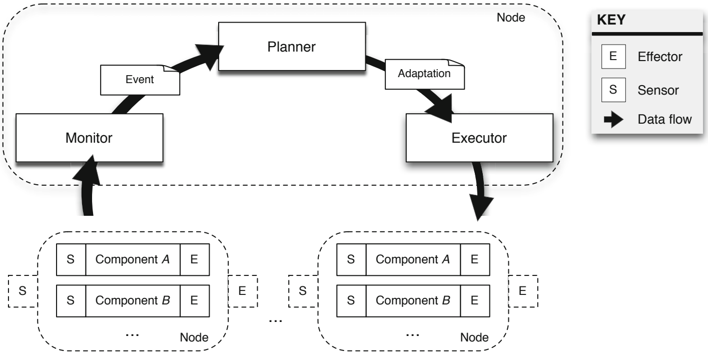


Fig. 2. Proposed architecture with the external controller (top node) and the managed system (bottom nodes)

sensors are associated to a component (e.g., a sensor that obtains the number of copies of an object that exists in the cache of Infinispan), while other sensors can be associated with a component instance (e.g., a sensor that measures the CPU use of a particular component instance). Sensors can also be associated to a node of the managed system. For instance a node-level sensor may measure the total energy consumed by a node, which depends on the components executing on that node but also on the OS, daemons, *middleware*, etc. By combining values from different sensors it is possible to derive KPIs of wider scope. For instance, by summing the CPU use of all components that execute in a given node it is possible to assess the total CPU utilization at that node. Data from multiple nodes may be aggregated to compute system-wide KPIs. For instance, energy consumed by each node may be added up to compute the total energy consumed by the entire cluster.

The monitor also analyzes the data and identifies system states that are undesirable and require correction. Whenever such a state is detected, the monitor notifies the planner, by triggering an event carrying the relevant state information. The monitor element is initialized with a set of event definitions that state in which conditions each event has to be triggered.

3.2 Planner

The *planner* determines how to adapt the system to meet its QoS requirements when it receives from the monitor an event signaling a deviation from the expected behavior. In the proposed approach, the result of the planner depends on the set of ranked behavior goals that have been defined for the system. This is because when the planner component searches for the best configuration for the system in the current state, it uses the notion of optimality defined by this set of goals and their ranks. On the other hand, the result of planner also depends on the set of mechanisms that are available for reconfiguring the system. It is assumed that the latter are specified in terms of adaptations that include an estimate of the impact they have in the system KPIs. When searching for the optimal configuration, the planner estimates the impact of different adaptations. Sections 4 and 5 explain the high-level goal-based planning of our approach in more detail.

3.3 Executor

The *executor* performs the adaptations selected by the planner. It relies on a set of commands that are different for each adaptation; these commands control the adaptation process. The commands are received and executed by effectors that are associated to adaptation targets. To support both distributed and non-distributed components, the effectors can be associated to a component or to a node. The former are often component-specific, thus, different effectors may need to be developed for each component. The latter are attached to a node and are able to configure the node.

To support the distribution, the executor needs to command both component and node effectors, but it also needs to be able to coordinate all the effectors while performing the selected adaptations. Furthermore, different adaptations may require different coordination mechanisms, which the executor must support. While the coordination aspects are fundamental to support the adaptations of distributed components, the discussion of these mechanisms in detail is outside the scope of this chapter. In this chapter we assume that all adaptations employ the barrier synchronization mechanism described in [15].

4 Adaptation Model

To control the self-management process, the planner relies on an adaptation model, which captures different key aspects related to the adaptive behavior and the adaptation support. As depicted in Figure 3, the adaptation model depends on a set of KPIs defined at design time. The system manager can express the target behavior for the system in terms of a high-level policy that establishes goals for some or all of these KPIs. When the system behavior deviates from the desired, its self-management is achieved using the adaptations described in the adaptation specification. The description of the components to which these adaptations apply is provided in the component specification.

4.1 KPIs

The general specification of a KPI includes its name, the type of value (integer, double, etc), and the error margin. Any two values of the KPI are considered equivalent if their distance is below this margin.

To address both distributed and non-distributed components, KPIs are divided in four categories that determine how their values are sensed: *system-sensed*, *component-sensed*, *node-sensed*, and *instance-sensed*.

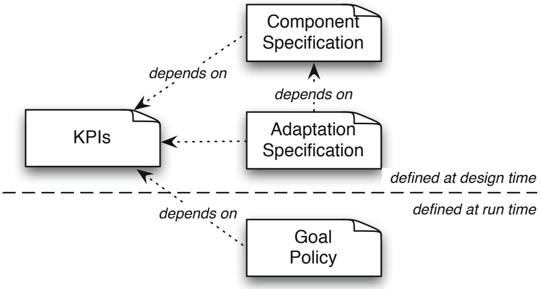


Fig. 3. The elements of the adaptation model

System-Sensed. The values of the KPI are measured for the entire system as a whole. For instance, the number of active servers in the system.

Component-Sensed. The values of the KPI are measured by individual component as a whole (even if they are distributed over different nodes) and its specification includes a *combination function CF*. This function defines how the KPI value for the entire system is obtained by combining the values measured for the individual components. For instance, the latency of requests may be measured by component and summed. We denote by *c.kpi* the value of *kpi* for component *c*.

Node-Sensed. The values of the KPI are measured by individual node and its specification includes an *aggregation function AF*. This function defines how the value of the KPI for the entire system is obtained by combining the values measured in each node of the system. This type of KPI is useful for metrics that are global to the node, thus, obtained independently of the components present in the node. For instance, the power consumption can be measured per node and these values can be aggregated taking the average. We denote by *n.kpi* the value of *kpi* for node *n*.

Instance-Sensed. The values of the KPI are measured per component instance, i.e., by individual component in each node. We denote by *n.c.kpi* the value of *kpi* for the component *c* in node *n*. The specification includes a *combination function* (CF) and an *aggregation function* (AF) that are employed to calculate the value of the global KPI from sets of values measured per component instance. The composition of the combination and aggregation functions is required to be commutative. The global value of an instance-sensed KPI is obtained by composing the two functions. In this way, it is possible to calculate the global value of the KPI in two manners:

a. $AF \circ CF$: first it is calculated the value of the KPI for each node through the application of the combination function, and then the values obtained for the different nodes are aggregated.
b. $CF \circ AF$: first it is calculated the value of the KPI for each component through the application of the aggregation function, and then the values obtained for the different components are combined.

For instance, the CPU use can be measured per component instance and we can take the sum as *combination function* and the average as *aggregation function* (it is not difficult to see that their composition is indeed commutative).

The fact that a KPI is component-sensed does not imply that its values have to be measured for every type of component. The KPIs that are measured for each type of component are defined in the component specification (see Section 4.4). The application of combination functions is, hence, restricted to the components for which these values are measured. More concretely, if a KPI *kpi* is component-sensed, its value for the entire system is given by the expression below, where $c < kpiMensurable$ denotes that the values of *kpi* are measured for c and $SysCfg()$ denotes the set of components in the current system configuration.

$$CF(\{c.kpi : c \in SysCfg() \ and \ c < kpiMensurable\})$$

Node-sensed KPIs are measured for all nodes. For instance, the power consumption of all nodes is measured, as idle nodes also consume power. The KPI value for the entire system is given by the following expression, where $\mathcal{N}$ denotes the set of the system's nodes.

$$AF(\{n.kpi : n \in \mathcal{N}\})$$

Combination and aggregation functions are required to be non-decreasing monotonic. For instance, for combination functions, this means that an increase in a component's KPI either maintains or increases the system's global KPI and, similarly, a decrease in one element either maintains or decreases it. This requirement expresses a natural property of KPIs and ensures that local reasoning about the type of impact that adaptations have in the KPI of individual components or nodes is also valid globally.

We provide below the definition of some KPIs useful in the context of the example application.

```
KPI–System number_servers:int Error 0
KPI–Comp latency:double CF Avg Error 0.1
KPI–Node power:int AF Sum Error 1
KPI–Instance cpu_use:double CF Sum AF Avg Error 0.1
KPI–Instance load:double CF Sum AF Sum Error 0.2
KPI–Instance throughput:double CF Sum AF Sum Error 0.2
```

KPIs can additionally be used to specify *composite* KPIs, denoted by CKPIs. CKPIs are identified by a *ckpi_name* and their specification consists of a join function *JF* of several KPIs, and an *Error* margin. In the example application, we can define the service ratio as a CKPI:

```
CKPI service_ratio = throughput/load Error 0.01
```

4.2 Goal Policy

The goals are the high-level directives which guide the system management. They describe the acceptable system behavior in terms of KPI values. A policy is a set of goals ranked by importance, with the most important coming first. When it is not possible to fulfill all the goals, the less important goals are violated first. There are six types of goals that can be defined.

```
Goal goalName:kpiName Above threshold_down
Goal goalName:kpiName Below threshold_up
Goal goalName:kpiName Between thr_down thr_up

Goal goalName:kpiName Close target MinGain value Every period
Goal goalName:Minimize kpiName MinGain value Every period
Goal goalName:Maximize kpiName MinGain value Every period
```

The first batch of goals are *exact goals*, which separate the values of a KPI in two disjoint sets: *acceptable* and *not acceptable*. An *above* goal will only find acceptable the values above the threshold. A *below* goal will only accept the values below, and a *between* goal only the values in the specified interval. The remaining goals are *optimization goals*, i.e., best effort goals that specify a total order between the values of a KPI. A *maximize* goal states that the largest is the best, while a *minimize* goal aims at the smallest. A *close* goal tries to keep the value as close as possible to a target. The description of optimization goals also specifies how often the system should try to optimize its behavior with respect to these goals (every *period* of time) and a *minimum gain* for an adaptation be worthwhile.

Below, we present two examples of goals for the example application. The first aims at maintaining the the system's redundancy level, by having at least three active servers. The second goal states that attempts of maximizing the service ratio should be done every 300 seconds. If an improvement is possible, the adaptation is only applied if the gain is 5% or more.

```
Goal preserve_redundancy:number_servers Above 3
Goal max_service_ratio:Maximize service_ratio MinGain 0.05 Every 300
```

4.3 Component Specification

The component specification includes the description of all the components available for use in the system. In particular, this specification defines a type hierarchy organizing components in types according to the functionality they provide. A component type is either concrete or abstract. The first designates a specific type for which an implementation is available, while the second represents the characteristics of a group of components. In the specification, a component is abstract if it is marked explicitly as *Abstract*, otherwise it is a concrete component. Both abstract and concrete components may describe their parents in the hierarchy by subtyping their types. The use of abstract components helps simplify both the component and adaptation specifications. The component specification becomes simpler because all parameters and subtypes of the parent component are inherited by children components. The adaptation specification also becomes simpler because adaptations that apply to all children components need to be specified only once, for the parent component.

For instance, in the example application, there are two types of catalog components tailored for private and business users. They can be specified as two subtypes of an abstract catalog component as follows.

```
Abstract Component Catalog
 subtype StoreService , {cpu_use,load,throughput,latency} Mensurable
 LocalParameters
  mode:{ regular , low}

Component BusinessCatalog
 subtype Catalog

Component PrivateCatalog
 subtype Catalog
```

The component specification also allows to describe which components contribute to instance and component-sensed KPIs. This is achieved by subtyping a corresponding abstract type *kpiMensurable*. For instance, *Catalog* is defined as a subtype of *cpu_useMensurable* and, hence, the value of *cpu_use* (which is a instance-sensed KPI) is available for every instance of the catalog.

When describing components, namely components that can be customized, their specification may also include the definition of parameters. These parameters can be tuned during runtime, with impact on the system behavior. There are two types of parameters that can be defined: *local* and *global*. Local parameters refer to parameters of a component instance or non-distributed components. Global components refer to parameters of the component as a whole.

The Infinispan component, specified below, is an example of a distributed component that has both local and global parameters. The *abortTimeout* is controlled per component instance, depending on the load and available bandwidth at each server, while the number of *owners* that an object has in cache (the number of copies) is a parameter for the component as a whole.

```
Component Infinispan
 subtype DistCaching ,{ cpu_use,load,throughput,latency} Mensurable
 LocalParameters
  abortTimeout : int
 GlobalParameters
  owners : int
```

4.4 Adaptation Specification

The adaptation specification describes the adaptations available to change the system's behavior. Adaptations are defined in terms of a fixed set of adaptation actions supporting the tuning of parameters, replacement of components and addition/removal of component instances. To address both distributed and non-distributed components, three groups of adaptations were considered, characterized by their scope: *instance*, *component*, and *node*.

Component adaptations employ actions that target a component *c*, affecting all the instances of that component in the system. There are two types of component adaptation actions: tune a parameter and exchange a component's implementation. For the first action, the *c.setParameter(p,v)* changes the value of the global parameter p of component c to v. For the second action, the *c.replaceBy(c')* replaces the implementation of c by c', affecting all nodes with instances of c. For instance, we can use the first action to change the global parameter *owners* of the *Infinispan* component, and the second action to update the version of the Infinispan component (that would affect all instances of Infinispan).

Instance adaptation employ actions that target a component c in a node n, affecting only that instance of c component in the system. There are two types of instance adaptation actions: tune a parameter and exchange a component's implementation. For the first action, the *n.c.setParameter(p,v)* changes the value of the local parameter p of component c to v in node n. For the second action, the *n.c.replaceBy(c')* replaces the implementation of component c by c' in node n. For instance, we can use the first action to change the *abortTimeout* parameter in a single node containing the Infinispan component and the second action to replace the implementation of the *User* component in a specific node.

Node adaptation employ actions that target only a node n. These actions change the node configuration and can affect the number of instances of distributed components. There are two types of node adaptations: add and remove a component. More concretely, the action *n.addComponent(c)* adds the component c to node n and *n.removeComponent(c)* removes the component c from node n. If the component is distributed, the first action adds an instance of c to node n and, hence, it changes the way c is distributed. For instance, these actions could be used to add/remove an instance of Infinispan to an inactive/active node.

The available adaptations to change the system's behavior are defined using the six types of adaptation actions defined above. Each adaptation, besides a set of adaptation actions, defines the conditions on its applicability, the impact of those actions on one or more KPIs and the estimated stabilization period. The impacts provide an estimate for the value of the affected KPIs after the specified adaptation actions have been performed. This estimate is expressed using an *impact function* that, besides the current value of the KPI, can be defined in terms of other KPIs or the state of the system configuration. The stabilization period refers to the time that must be waited for the adaptation to take full effect, before subjecting the system to a reevaluation.

Below, we present a simplified example of each adaptation type for the example application, with the full specification being addressed in Section 6.

```
ComponentAdaptation ActivateTotalOrder(c)
 Component:
  c:JGroups
 Actions:
  c.setParameter(totalOrder,on)
 Requires:
  c.totalOrder == off
 Impacts:
  latency *= 1.07
  ...
 Stabilization:
  period = 60 secs

InstanceAdaptation IncreaseAbortTimeout(n,c)
 Node:
  n
 Component:
  c:Infinispan
 Actions:
  n.c.setParameter(abortTimeout,n.c.abortTimeout+10)
 Requires:
  n.c.abortTimeout < 60
 Impacts:
  n.c.latency *= 1.1
  ...
 Stabilization:
  period = 60 secs

NodeAdaptation AddServer(n)
 Node:
  n
 Actions:
  n.addComponent(ApacheHTTP)
  n.addComponent(Infinispan)
  ...
 Requires:
  ! n.hasComponent(ApacheHTTP)  //if n is a free node
 Impacts:
  number_servers += 1
  latency ÷= 1.89
  n.power += 183.1
  ...
 Stabilization:
  period = 60 secs
```

Adaptations can declare to have impact on any KPI that has the same or larger scope. In particular, any adaptation may declare an impact on a system-sensed KPI. Ideally, the impact of an instance adaptation on a instance-sensed KPI *kpi* should be defined at the instance-level, i.e., it should address the change of *n.c.kpi* (where *c* and *n* are the target component and node of the adaptation). However, because it is not always possible or easy to find such fine-grained impacts, it is also possible to define the impact of an instance adaptation at higher-levels, i.e., at component, node or even system level. And, in a similar way for the other two types of adaptations. For instance, the node adaptation *AddServer* specifies impacts on the *number_servers* system-sensed KPI and on the value of *n.power*, which is a node-sensed KPI. Moreover, the adaptation also specifies impact on the system value of the *latency*, which is a component-sensed KPI. Although the adaptation targets a single node, it affects the *latency* in all nodes and, hence, the impact is defined at the system-level rather than at the node-level.

In addition to the information above, the adaptation specification may also list dependencies and explicit conflicts between pairs of adaptations, to force or prevent that these adaptations are executed at the same time. However, some conflicts are also imposed by our approach, hereafter referred to as implicit conflicts. An implicit conflict occurs when it is not possible to infer the combined impact of two or more adaptations from the information provided in their specification. More concretely, any pair of adaptations to which the following scenarios apply are considered to be implicitly conflicting:

- Instance adaptations that affect the same component, the same node or both, and have impact over a common KPI;
- Component or instance adaptations that affect the same component and have impact over a common component or system-sensed KPI;
- Node or instance adaptations that affect the same node and have impact over a common node or system-sensed KPI;
- Component and instance adaptations that have impact on different components but on the same node-sensed KPI (the first on the global value, while the second on the local node value).
- Component and node adaptations that have impact on the same KPI.

5 Rule Generation and Evaluation

With the knowledge and information necessary for self-management described in the adaptation model, the planner can determine when and how to adapt the system behavior. As shown in Figure 4, the activity of the planner encompasses an offline and an online phase.

During the *offline* phase, the knowledge and information described in the adaptation model is used to generate a set of rules. Each rule defines a set of adaptations that might help to correct a particular deviation in the system behavior. The rule generation process does not depend on the system configuration, thus, can be performed at any time, in particular before system execution.

During the *online* phase, the rules generated for the system are evaluated. When the system deviates from the desirable behavior, the corresponding rule is triggered and it is determined the most appropriate manner to return the system to a desired state.

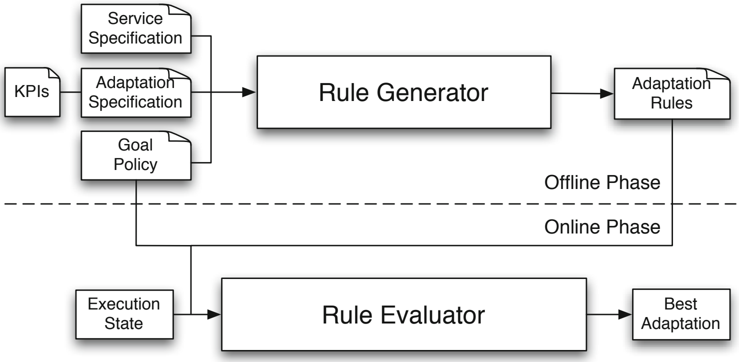


Fig. 4. Rule generation and evaluation

5.1 Offline Phase

During this phase, the knowledge and information described in the adaptation model is used to generate a set of event triggered rules, where each event signals a particular deviation in the system behavior with respect to the system goals. Hence, the rule generation starts by extracting the set of relevant events from the goals included in the high-level policy. Each goal results in either one or two generated events. For instance, the *preserve_redundancy* goal results in the event *kpiBelow(num_servers,3)*, that is triggered when *number_servers* is below 3. Table 1 describes which events are generated per type of goal. Exact goals give rise to events triggered whenever a threshold is exceeded, while optimization goals result in periodic events, triggered every period of time defined in the goal.

Table 1. Event types generated for each type of goal

Goal	**Events**
k Above x	kpiBelow(k, x)
k Below y	kpiAbove(k, y)

k Between x y	kpiBelow(k, x)
	kpiAbove(k, y)

Maximize k Every θ	kpiIncrease(k, θ, true)
Minimize k Every θ	kpiDecrease(k, θ, true)

k Close x Every θ	kpiIncrease(k, θ, "$<x - k_{error}$")
	kpiDecrease(k, θ, "$>x + k_{error}$")

After extracting the events from the goals in the policy, the rule generation proceeds with the identification of the useful adaptations for each event. An adaptation is useful if it has impact on the KPI or KPIs (if it is a CKPI) that are employed in the goal and if that impact helps returning the KPI to a desirable state. Although the actual KPI value, after the adaptation, is only known at runtime, it is possible to filter out the non useful adaptations. An adaptation is not useful in two situations: it does not have impact on the KPIs associated with the goal or, if it has, the impact is negative. An impact is negative if the adaptation changes the KPI's value in a direction contrary to the desired. For instance, if the value of a KPI has become larger than desired, a useful adaptation decreases the KPI value. The adaptations that decrease the KPI have a positive impact, while the ones that increase have a negative impact. To determine offline if an impact is negative or positive the approach relies on the analysis of the impact function. If this analysis is not able to classify an impact as positive or negative, it is selected nonetheless, to avoid discarding useful adaptations.

The adaptations that are considered in this process include the instantiation of all the node and instance adaptations that have been defined as parameterized with the actual system nodes and concrete components. For instance, if there are ten nodes available for running the servers, then the instantiation of the parameterized adaptation *addServer* would give rise to ten different adaptations.

While in some scenarios a single adaptation may be enough to correct the system behavior, in other cases putting together several adaptations is necessary. Thus, the rule generation process not only selects the adaptations but also determines the viable combinations of adaptations, i.e., adaptations that can be performed at the same time. This process considers any requirements, dependencies and conflicts between adaptations. In the end of the offline phase, there are several generated rules, one for each extracted event. Each rule associates a set of combinations of adaptations to a particular event, as depicted below, where C_i is a viable combination of adaptations useful to deal with an *event*.

```
When event
  Select  C1 , . . . , Ck
```

When it is necessary to change the goal policy during runtime, the offline phase takes place during runtime, in parallel to the online phase. The generation of the rules is performed in the same manner as if before runtime, without causing any disruption to the self-management. However, at the end of the offline phase, it is necessary to replace the rules used by the online phase and feed the monitor with the events generated from the new goal policy. This can be done at any time, except when the online phase is evaluating a rule. If this is the case, the rule replacement will have to wait until the end of the evaluation.

5.2 Online Phase

In this phase, the rules generated during the offline phase are evaluated. When the system reaches an undesirable state, the rule triggered by the corresponding event is evaluated. The aim of this evaluation is to select, among the combination of adaptations available in the rule, the combination that is optimal with respect to the ranking of goals in the policy.

Using as input the current system state (available in the monitor), the evaluation process starts by discarding all the adaptations in the rule that do not apply to the current system configuration. For the remaining adaptations, the KPI values are estimated using the impact functions. Afterwards, the selection algorithm analyzes the combinations of adaptations with respect to the each goal (starting with the highest rank goal) until all the goals are addressed or until no more adaptations are left. If a combination fulfills a gigen goal, it passes to the next step, otherwise, it is discarded. The selection algorithm is sensitive to the type of goal. If the goal is an exact goal, all the combinations of adaptations whose estimated impacts satisfy the goal are selected. If none of the combinations satisfies the goal, then all the adaptations continue to the next goal. If the goal is an optimization goal, then only the best adaptations are selected. The best adaptations are those that offer the best improvement to the KPI.

This selection process of the combination of adaptations is optimal with respect to a prioritization of goals based on their rank: when it is not possible to fulfill all goals, the rule evaluation process will enforce that goals with lower rank are violated first. A more detailed description of the selection criteria can be found in [16,17] along with some detailed examples.

6 Example Revisited

In this section, we illustrate the use of the approach by applying it to the example application introduced in Section 2. We present the full description of the example application: the KPIs, the component and adaptation specifications, and the goal policy. While the KPIs and the specifications allow for many adaptive behaviors, the goal policy describes one adaptive behavior from the various alternatives. In this chapter, we focus on adaptations and trade-offs related with the distributed aspects of the application. Therefore, the goals listed in the policy described here are a subset of the goals that would cover the entire adaptive behavior.

6.1 KPIs

To assess the system's behavior, we rely on several KPIs and one CKPI as listed, respectively, in Tables 2 and 3. Since some of the KPIs have already been described in Section 4.1, we only address in detail the newly introduced KPIs.

Table 2. KPIs used in the example application

Type	KPI Name	Values	CF	AF	Error	Description
System	number_servers	int	-	-	0	active servers
Component	latency	double	Avg	-	0.1	reply delay
Component	update_ratio	double	Avg	-	0.1	fraction of update transactions
Node	power	int	-	Sum	1	power consumption
Instance	cpu_use	double	Sum	Avg	0.1	cpu consumption
Instance	load	double	Sum	Sum	0.2	incoming reqs/s
Instance	throughput	double	Sum	Sum	0.2	processed reqs/s
Instance	fidelity	integer	Sum	Sum	1	content quality
Instance	abort_ratio	double	Avg	Avg	0.0001	aborts per second

Table 3. CKPI used in the example application

CKPI Name	Values	JoinFunction	Error	Description
service_ratio	[0,1]	throughput/load	0.01	service ratio

The only system-sensed KPI — *number_servers*, represents the number of deployed servers in the system. There are two component-sensed KPIs: *latency*, and *update_ratio*, that describes the fraction of update transactions that take place in the cache tool. There is a single node-sensed KPI — *power* consumption and five instance-sensed KPIs: *cpu_use*, *load*, *throughput*, *fidelity*, and *abort_ratio*. The *fidelity* level represents the quality of the served content by each of the non-distributed components. The *abort_ratio* in Infinispan represents the aborted transactions per second. The caches in Infinispan keep dynamic data used to generate webpages, but this data may change. For instance, when a user updates their profile, in addition to updating the database, the cache must be updated too. When two requests try to write on the same block of data, a conflict happens and the operations are aborted. Finally, there is a single CKPI — *service_ratio*, that indicates how adequate the capacity is to the current load.

6.2 Component Specification

The example application makes use of twelve components. The final and complete specification is presented next, including the descriptions already presented in Section 4.3. Since the specification of the business and private Catalog, User, and Account components is similar, they are described together.

```
Abstract Component StoreService
 subtype {cpu_use,load,throughput,latency} Mensurable

Abstract Component Catalog/Account
 subtype StoreService ,fidelity Mensurable
 LocalParameters
  mode:{ regular ,low}

Abstract Component User
 subtype StoreService

Component BusinessCatalog/User/Account
 subtype Catalog/User/Account

Component PrivateCatalog /User/Account
 subtype Catalog/User/Account

Abstract Component DistCaching

Component Infinispan
 subtype DistCaching ,{ cpu_use,load,throughput,latency,abort_ratio,update_ratio} Mensurable
 LocalParameters
  abortTimeout : int
 GlobalParameters
  owners : int

Abstract Component Multicast

Component JGroups
 subtype Multicast ,{ cpu_use,latency} Mensurable
 GlobalParameters
  totalOrder :{ on , off }
  number_caches : int

Abstract Component WebServer

Component ApacheHTTP
 subtype WebServer ,{ cpu_use,latency} Mensurable

Abstract Component Engine
 subtype {throughput,latency,fidelity} Mensurable
 LocalParameters
  mode:{ fresh , cache}

Component Searches
 subtype Engine

Component Recommendations
 subtype Engine
```

In the specification above there are several noteworthy aspects. One is that not all components contribute to the same KPIs. For instance, in terms of *load* and *throughput* only the non-distributed components and the caching tool contribute, because the web server works as a relay, and the multicast service does not actually process web server requests. Another noteworthy aspect is that non-distributed components only declare local parameters. This avoids incorrect adaptations in scenarios where there are several instances of non-distributed components running in different nodes. By marking the parameter as local, the executor will not try to adapt the parameter in all instances, but only on the targeted node. Finally, another noteworthy aspect is that the use of abstract components simplifies the specification, avoiding repeating the subtypes and parameters for children components.

6.3 Adaptation Specification

In this section, we describe the final specification of all adaptations. Six *reversible* and two regular adaptations have been defined for the system. Each *reversible* adaptation, gives rise to two adaptations — the specified adaptation and its inverse adaptation. These adaptations can be combined as long as there are no implicit conflicts. The adaptation specification does not include explicit conflicts nor dependencies, as the conflicts between adaptations are already covered by the implicit conflicts.

```
Reversible NodeAdaptation AddServer(n)
 Node:
  n
 Actions:
  n.addComponent(ApacheHTTP)
  n.addComponent(Infinispan)
  n.addComponent(JGroups)
  n.addComponent(BusinessCatalog)
  n.addComponent(PrivateCatalog)
  n.addComponent(BusinessUser)
  n.addComponent(PrivateUser)
  n.addComponent(BusinessAccount)
  n.addComponent(PrivateAccount)
 Requires:
  ! n.hasComponent(ApacheHTTP)
 Impacts:
  number_servers += 1
  latency ÷= 1.89
  n.power += 183.1
  throughput = 1/(update_ratio*
      (1−abort_ratio)*cacheWriteTime)
 Stabilization:
  period = 100 secs

ComponentAdaptation IncreaseOwners(c)
 Component:
  c:Infinispan
 Actions:
  c.setParameter(owners, c.owners+1)
 Requires:
  c.owners < c.number_caches
 Impacts:
  c.throughput *= 1.23
  c.latency *= 1.12
 Stabilization:
  period = 120 secs

ComponentAdaptation DecreaseOwners(c)
 Component:
  c:Infinispan
 Actions:
  c.setParameter(owners, c.owners−1)
 Requires:
  c.owners > 1
 Impacts:
  c.throughput ÷= 1.23
  c.latency ÷= 1.12
 Stabilization:
  period = 120 secs

Reversible InstanceAdaptation
    CatalogToLow(n,c)
 Node:
  n
 Component:
  c:Catalog
 Actions:
  n.c.setParameter(mode, low)
 Requires:
  n.c.mode == regular
 Impacts:
  n.c.latency ÷= 0.03
  n.power ÷= 0.007
  n.c.cpu_use ÷= 2.01
  n.c.throughput *= 1.78
  n.c.fidelity −= 1
 Stabilization:
  period = 60 secs

Reversible InstanceAdaptation
    AccountToLow(n,c)
 Node:
  n
 Component:
  c:Account
 Actions:
  n.c.setParameter(mode,low)
 Requires:
  n.c.mode == regular
 Impacts:
  n.c.latency ÷= 1.02
  n.power ÷= 0.004
  n.c.cpu_use ÷= 1.65
  n.c.throughput *= 1.38
  n.c.fidelity −= 1
 Stabilization:
  period = 60 secs

Reversible InstanceAdaptation
    IncreaseAbortTimeout(n,c)
 Node:
  n
 Component:
  c:Infinispan
 Actions:
  n.c.setParameter(abortTimeout,n.c.
      abortTimeout+10)
 Requires:
  n.c.abortTimeout < 60
 Impacts:
  n.c.latency ÷= 1.1
  n.c.throughput *= 1.38
  n.c.abort_ratio ÷= 2.9
 Stabilization:
  period = 60 secs


Reversible ComponentAdaptation
    ActivateTotalOrder(c)
 Component:
  c:JGroups
 Actions:
  c.setParameter(totalOrder,on)
 Requires:
  c.totalOrder == off
 Impacts:
  latency *= 1.07
  throughput *= (11−update_ratio
   * log(1.2*number_servers))
  abort_ratio ÷= 10.7
 Stabilization:
  period = 60 secs

Reversible InstanceAdaptation
    FreshGenerations(n,c)
 Node:
  n
 Component:
  c:Engine
 Actions:
  n.c.setParameter(mode,fresh)
 Requires:
  n.c.mode == cache
 Impacts:
  latency *= 1.34*(PrivateUser.load +
      BusinessUser.load)
  throughput ÷= 1.4
  n.c.fidelity += 1
 Stabilization:
  period = 60 secs
```

In the specification of the adaptations there are some aspects worth noting. The impacts of the adaptations were derived from experimentation and benchmarking, and their description may rely on functions employing other KPIs and constants. For instance, in the *IncreaseAbortTimeout* adaptation, the description of the impact on the throughput not only uses the *update_ratio* KPI but also a constant that is the *cacheWriteTime* — that describes the average time that an update will take in a machine with some specific hardware characteristics. Another noteworthy aspect is the use of parameters, which make the adaptation specification simpler. The adaptations *CatalogToLow*, *AccountToLow*, and *FreshGenerations* are described for components of abstract type. As a result, instead of describing the six reversible adaptations, only three reversible adaptations are described, one for each abstract component.

The specification for the example application does not include any dependencies or conflicts. While there are conflicts between some adaptations, the implicit conflicts already rule out the pairing of those adaptations. For instance, the *AddServer* adaptation and its inverse have an implicit conflict because both have impact on the *number_servers* KPI, therefore, it is not necessary to specify that explicitly.

6.4 Policy

As previously mentioned, in this chapter we only discuss the portion of the policy that address the aspects related with distribution of components. The high-level goal policy described in this chapter aims at an adaptive behavior that takes advantage of the trade-offs related with distribution. Even by narrowing down the policy to a sub-set of the possible goals, there are still several alternatives policy specifications. We illustrate the approach using the following policy:

```
Goal preserve_redundancy: number_servers Above 3
Goal limit_abort_ratio: abort_ratio Below 0.008
Goal max_service_ratio: Maximize service_ratio MinGain 0.05 Every 300
Goal min_latency: Minimize latency MinGain 0.05 Every 400
Goal save_resources: Minimize number_servers MinGain 0 Every 500
```

The policy consists of five goals. The first goal, which is the most important, is to maintain the redundancy, i.e., a minimum number of servers to process requests. This self-healing property is the most important goal because it will allow the system to recover from fail overs and avoid downgrading the service to a critical level. The next three goals address performance issues. The *limit_abort_ratio* is ranked as second because an abort ratio higher than the defined threshold (0.008) renders the system irresponsive due to being blocked most of the time (we have determined this threshold experimentally, through benchmarks). The third goal aims at processing as many requests as possible, i.e, to maximize the service ratio. The fourth goal is to minimize the latency, which is considered less important than the service ratio. Finally, the last goal minimizes the resource consumption.

Table 4. Events extracted from the goals

Type	Goal	Event	Trigger
Exact	*preserve_redundancy*	kpiBelow(*number_servers*,3)	*number_servers*< 3
Exact	*limit_abort_ratio*	kpiAbove(*abort_ratio*,0.008 + 0.0001)	*abort_ratio*> 0.0081
Approx	*max_service_ratio*	kpiIncrease(*service_ratio*,300,true)	Every 300 s
Approx	*min_latency*	kpiDecrease(*latency*,400,true)	Every 400 s
Approx	*save_resources*	kpiDecrease(*number_servers*,500,true)	Every 500 s

From the goal policy above, five events were extracted, as depicted in Table 4. Then five adaptation rules are generated, taking into account that the system runs in ten nodes numbered from 1 to 10 plus a node designated by *backendNode*. Next, we show some of the combinations present in each rule.

```
When kpiBelow(number_servers ,3)
 Select {AddServer(1)} ,...

When kpiAbove(abort_ratio ,0.0081)
 Select {ActivateTotalOrder(JGroups)},
        {IncreaseAbortTimeout(1,Infinispan)},...

When kpiIncrease(service_ratio,300,true)
 Select {ActivateTotalOrder(JGroups)},
    {IncreaseAbortTimeout(1,Infinispan)},...
        {FreshGenerations_Inverse(1,Searches)},...,
        {CatalogToLow(1,BusinessCatalog)},...,
        {CatalogToLow(1,PrivateCatalog)},...,
        {AccountToLow(1,BusinessAccount)},...,
        {AccountToLow(1,PrivateAccount)},...,
        {AddServer(1)},...

When kpiDecrease(latency,400,true)
 Select {ActivateTotalOrder_Inverse},
        {FreshGenerations_Inverse(1,Recommendations)},...,
        {AddServer(1)},...

When kpiDecrease(number_servers,500,true)
 Select {AddServer_Inverse(1)},...
```

6.5 Prototype

We have built a proof-of-concept prototype implementation of the approach in the Java$^{\mathrm{TM}}$programming language. The prototype includes the self-management elements and the example application. In terms of self-management elements, the monitor, planner and executor are run in a dedicated node. The database, NAS and engines are also run in a dedicated node. The remaining nodes run servers of the example application.

The implementation of the self-management support is done entirely using Java$^{\mathrm{TM}}$objects. The specification of components and adaptations is achieved by declaring component and adaptation interfaces. The KPIs and CKPIs used to characterize the system are described through the creation of a set of KPI and CKPI objects in the monitor element. The goal policy is written in the planner through the creation of a list of goal objects. The rules generated from the goal policy and adaptation specifications are described using a set of rule objects.

The monitoring of the system and environment is performed by the monitor and all the sensors associated to components. Each component has a dedicated sensor that captures the information regarding the *mensurable* KPIs described in the component specification (see Section 4.3). Each sensor is designed for a particular component, as different component provide different KPI information and have different mechanisms to access that data. To implement some sensors it was necessary to alter the component implementation, as the data was not available to elements external to the component. This is the case of the Infinispan and JGroups components.

The execution of the adaptations selected by the planner is performed by the executor element and all the effectors associated to the components and nodes. The implementation of the executor highly depends on the supported adaptations, and the targeted system and components. The execution of an adaptation may need to take many aspects in consideration, such as achieving quiescence, transferring state, or coordinating the instances involved in an adaptation, among many others. To address any of these concerns there are several alternatives available, some addressed in [15]. The executor developed for this prototype and example application is prepared to handle quiescence and node coordination. The quiescent state is a responsibility of the component, while the coordination is of the executor and effectors. The executor is prepared to perform both non-distributed and distributed adaptations. The distributed adaptations are performed without the need for synchronization barriers, which speeds up the adaptation execution and minimizes the disruption caused to the system execution. This is possible due to dedicated mechanisms in JGroups, that help the effectors executing the adaptation without interrupting the service.

6.6 Evaluation

The prototype uses a cluster of eleven machines with the following configuration: Dual Intel Xeon Quad-Core, 2.13 GHz clock speed, and 8 GB of RAM running Linux (kernel 2.6.32-21-server). All machines are connected by a 1 Gbps Ethernet. We used Infinispan version 5.0.0, extended with a number of sensors and effectors. Infinispan was deployed in full replication mode (i.e., each data item is replicated in every active server). Furthermore, we have used JGroups 2.11.0, which has been also augmented with an effector that is able to activate or deactivate the total order layer of the JGroups group communication stack. The workload is emulated by Radargun benchmark [14], version 1.1.0. The benchmark simulates the clients, the virtual server load balancer, and the web servers at each node. The benchmark is able to detect when a new instance of the cache is added to the cluster, through a monitoring agent present in each node.

To illustrate the operation of our autonomic manager and the effect of the high-level goal policy, a deployment of the system was subject to a variable load. All experiments follow the same pattern: we first let the system stabilize in the best configuration for a given workload, then we change the workload characterization and observe how the system reacts. Changes to the workload are made such that different adaptations are more appropriate in each experiment. The workloads are differentiated by two main characteristics: *high contention* (HC) or *low contention* (LC), and *load*. The high contention HC-3 workload captures a scenario where concurrent accesses to the same item occur often, which creates many opportunities for deadlocks and a potential increase in the abort ratio. This workload is tailored for the capacity of three servers. There are three low contention workloads: LC-3, LC-5, and LC-6. They require different capacities to handle the load, respectively, 3, 5 and 6 servers. Using this set of workloads, we have experimented 4 different transitions, namely: i) LC-3 to HC-3; ii) HC-3 to LC-3; iii) LC-5 to LC-6; and iv) LC-6 to LC-5. The results obtained for each transition are depicted in Figure 5.

In the first scenario, LC-3 to HC-3, the workload changes from low contention to high contention. When operating with low contention, JGroups is running without total order, as it allows to obtain a lower latency. When the workload changes there is a significant increase in the abort ratio, which degrades the service ratio, as fewer requests are served with success. This degradation of the service ratio can be observed in Figure 5(a) until the vertical line (that marks the adaptation).

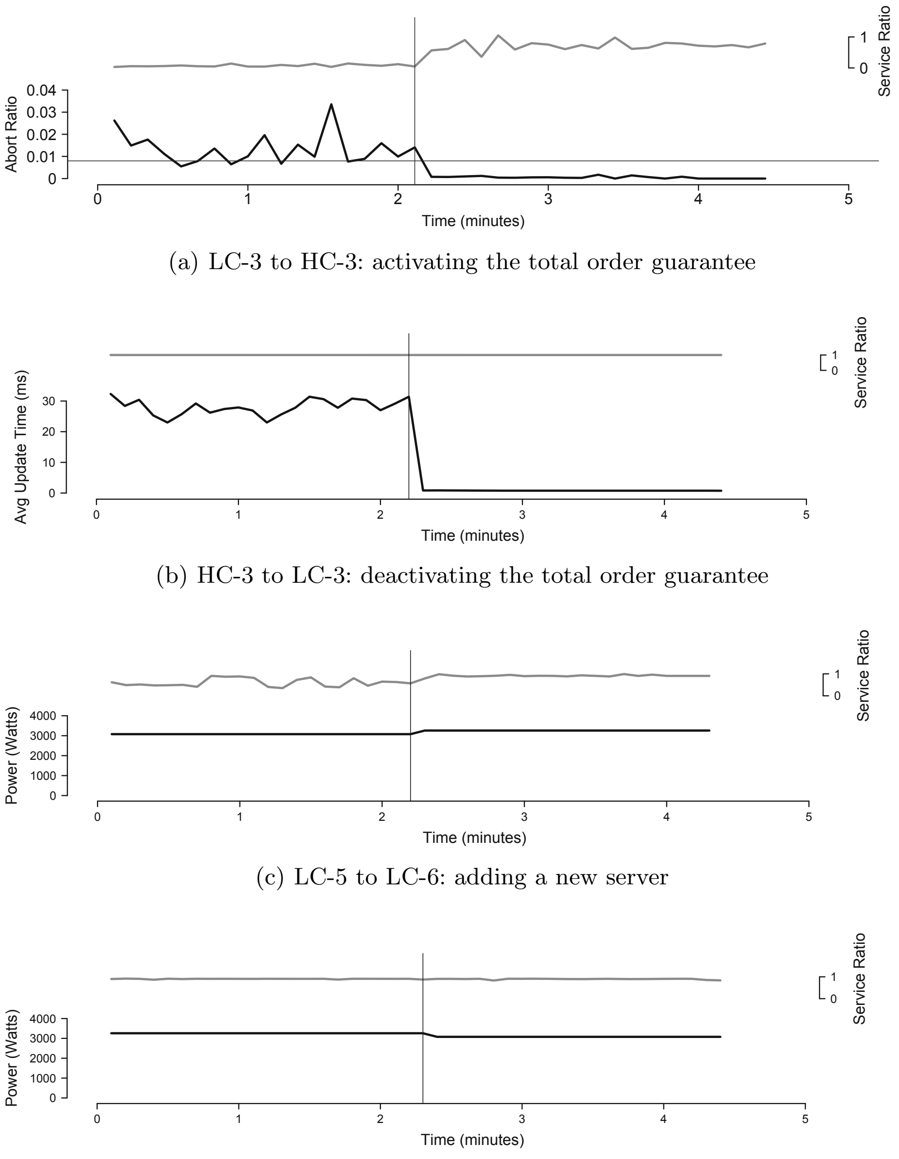
As the graphic shows, the abort ratio violates the *limit_abort_ratio* goal, because it is above the threshold specified in the policy. In the planner, the corresponding rule is evaluated, using the current system state, carried by the event. The selected adaptation is to *ActivateTotalOrder* guarantee. As a result, the abort ratio decreases and, consequently, the service ratio increases, which can also be observed in Figure 5(a), after the adaptation, when the system stabilizes.

The second scenario, HC-3 to LC-3, illustrates the inverse adaptation. The transition of workloads results in a decrease of the contention level and of the abort ratio, allowing room to improve the latency. The planner determines that it is possible to improve the latency by using the *ActivateTotalOrder_Inverse* adaptation (deactivates the total order), without compromising any of the higher ranked goals. Figure 5(b) shows that not only the service ratio is not degraded by the adaptation, but the update latency is reduced. This effect is mostly noticeable by observing the average update time (available as context information).

In the third scenario, LC-5 to LC-6, the system is operating under a low contention scenario, such that total order is not required, and the load is efficiently handled by 5 servers. With the transition of workloads, the service ratio drops due to the system overload. The planner element selects the *AddServer* adaptation to increase the system capacity, thus, increasing the rate of served requests and, consequently, the service ratio. Figure 5(c) depicts the service ratio before and after the adaptation, showing a clear improvement of the service ratio, returning its value back to 1. However, this adaptation results in the system demanding more power, since the newly active server is no longer idle, as Figure 5(c) also depicts. We opted to show the average power consumption because power consumption is not steady over time.

The final scenario, LC-6 to LC-5, is the inverse of the previous scenario. After the transition, the 6 servers are no longer required to maintain a service ratio close to 1, with resources being unnecessarily consumed by the sixth server. The planner will select one *AddServer_Inverse* adaptation (that removes a server) to decrease the resource consumption. Figure 5(d) shows that after the adaptation, the service ratio is maintained, still efficiently processing all incoming requests, and power is saved.

The main goal of the prototype was to demonstrate that that our approach is viable and useful in the context of distributed systems. The results show that, in distinct scenarios, the proposed approach is able to automatically guide the system adaptation during runtime according to the management goals. As we have seen, this involves balancing the different trade-offs and selecting the adaptation that best satisfies these goals. Although we have only considered a sub-set of the possible goals, the number of potential configurations of the system is large. Therefore, it is not easy to devise sets of adaptations that adequately balance these trade-offs while, at the same time, have a good coverage for the possible range of the system's operational conditions. Larger sets of adaptations, KPIs, and goals dramatically increase the difficulty of this task and also pose



(a) LC-3 to HC-3: activating the total order guarantee

(b) HC-3 to LC-3: deactivating the total order guarantee

(c) LC-5 to LC-6: adding a new server

(d) LC-6 to LC-5: removing one server

Fig. 5. Experimental results some minutes before and after the adaptation

scalability issues for the proposed approach. We have addressed these issues in [17] through an evaluation study of the approach's performance and scalability, that shows that the approach scales well up to 400 adaptations.

7 Related Work

In the proposed approach, the planning activity relies on high-level goal policies. When compared to other approaches in which the adaptation is based on a portfolio of adaptation strategies (defined at design time), this choice has several advantages. Approaches that employ goal policies are often better equipped to address dynamic environments with some level of uncertainty. If the adaptation is exclusively guided by adaptation strategies, the occurrence of changes not anticipated (with no adaptation strategy associated), leaves the system without reaction. Another important advantage is that it also deals better with system evolution. The addition of new components or more adaptations to existing components can be done while maintaining the goal policy. The flexibility to change the system's adaptive behavior is also better, given that this can be achieved by changing the goal policy. In what follows, we focus on approaches that also employ high-level goals to achieve the system's adaptive behavior.

There are several policy-based approaches that employ *high-level* or *abstract* goals to address the drawbacks posed by low-level action policies. However, we are not aware of other approaches (or approaches with the means to be easily extended) that support the adaptation of systems with distributed components.

Approaches that employ goal policies usually rely in some form of mapping between goal policies and action policies. The approach in [2] relies on a direct mapping from goal policies expressed in temporal logic to action policies. Not only the system designer needs to define all the behavior goals and the possible adaptation actions for the system, but she must also manually describe the link between goals and actions. The goal *policy refinement* then relies on abductive reasoning to find which sequence of operations allows to achieve the goals. However, neither optimization goals, nor graceful degradation are supported. Furthermore, the goals only address component properties, excluding any global system properties. Other approaches like [18] address global system properties, relying instead on reactive system techniques to perform the selection in a more systematic manner. However, they do not provide the means to derive the correct value of a parameter in a set parameter adaptation. Finally, both approaches cannot change the goal policies during runtime.

In the three-layer reference model [19], the goal policy is also expressed in terms of temporal logic formulas. These goals, together with a description of the system capabilities, are used to generate action policies to enforce the goals. This generation relies in identifying all the states from which it is possible to lead the system to a correct state, thus, creating a rule for each undesired but amendable state. This approach suffers from the same issues mentioned in the previous approach. Nonetheless, it allows changes to the goals during runtime. Both approaches lack the ability to balance conflicting goals, therefore, the trade-offs of performing an action.

Goal policies have been also specified using utility functions [20,9,6,8], in alternative to temporal logic or ranked goals. This is achieved by defining the utility (a scalar value) of each possible system configuration as a function of specific data available in the context (e.g., memory and bandwidth available). Hence, it demands that the user is able to quantify the importance of each goal and find a delicate balance for all the goals. In these approaches, the aim is to assemble a configuration tailored to the current situation. For instance, in MADAM [9], a goal policy is expressed in terms of an utility function that assigns a scalar value to each possible system variant, as a function of the system properties in a given context. When the system needs to be adapted, the choice of a system variant relies on property predictor functions over the associated system properties.

Predictors of the impact of adaptations decisions on the system's goals are used in several approaches to self-adaptation, namely they are also used in [6,8]. In Stitch [6], an adaptation tactic is associated with an impact vector, which determines its expected contribution to the system utility dimensions while Fusion [8] adopts a learning based approach in which the analytical model that relates the impact of adaptation decisions on the system's goals is automatically induced from the monitored data. The learning techniques developed in the context of Fusion are in fact an excellent way of overcoming potential imprecisions of the impact functions that, in our approach, have to be defined at design-time. In particular, they would promote the definition of rough approximations for these functions with obvious advantages in terms of required time and expertise.

8 Conclusions and Final Remarks

Building distributed systems that are able to efficiently cope with dynamic and unpredictable environments and workloads is still a challenge today. In this chapter, we propose a self-management approach designed for systems with both distributed and non-distributed components that are customizable. The self-management relies on a high-level goals policies, described by system managers, that may express an SLA, performance and resource consumption concerns, among other considerations. For that purpose, the approach relies on a number of KPIs to establish goals for the system behavior. The approach also leverages the knowledge of component developers, namely, information on how the components can be adapted and the adaptation trade-offs in terms of KPIs. The runtime monitors the system and is able to automatically select and deploy the necessary adaptations to correct deviations in behavior. We have validated the proposed approach using a prototype of a high traffic web cluster that employs both distributed and non-distributed components.

In the future, we plan to extend our work in several directions. First, we would like to allow the maximum number of nodes to be dynamic, where the number of available nodes changes during runtime. This support demands that the unfolding of instance and node adaptations is performed during runtime. Second, we would like to incorporate the cost of performing an adaptation in the selection algorithm. In the distributed setting this is particularly interesting, as the cost of an adaptation that involves several instances of a component is usually larger than a single instance. Therefore, if there are several possible adaptations, it would be preferable to choose the one with the lowest cost.

Acknowledgements. This work was partially supported by the FCT (INESC-ID and LASIGE multi annual funding through the PIDDAC Program fund grant and by the project CMU-PT/ELE/0030/2009).

References

1. Apache, http://httpd.apache.org
2. Bandara, A.K., Lupu, E.C., Moffett, J., Russo, A.: A goal-based approach to policy refinement. In: IEEE International Workshop on Policies for Distributed Systems and Networks, p. 229 (2004)
3. Bridges, P.G., Hiltunen, M.A., Schlichting, R.D.: Cholla: A framework for composing and coordinating adaptations in networked systems. IEEE Trans. Comput. 58, 1456–1469 (2009)
4. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
5. Cheng, B.H., Lemos, R., Giese, H., Inverardi, P., Magee, J.: Software Engineering for Self-Adaptive Systems. Springer, Heidelberg (2009)
6. Cheng, S.W., Garlan, D., Schmerl, B.: Architecture-based self-adaptation in the presence of multiple objectives. In: Proceedings of the 2006 International Workshop on Self-adaptation and Self-managing Systems, pp. 2–8. ACM (2006)
7. Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A survey of autonomic communications. ACM Trans. Auton. Adapt. Syst. 1, 223–259 (2006)
8. Elkhodary, A., Esfahani, N., Malek, S.: Fusion: a framework for engineering self-tuning self-adaptive software systems. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 7–16. ACM (2010)
9. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using architecture models for runtime adaptability. IEEE Softw. 23(2), 62–70 (2006)
10. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37(10), 46–54 (2004)
11. Infinispan, http://www.jboss.org/infinispan
12. JGroups, http://www.jgroups.org/
13. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36, 41–50 (2003)
14. Radargun: See, http://sourceforge.net/apps/trac/radargun/
15. Rosa, L., Rodrigues, L., Lopes, A.: A framework to support multiple reconfiguration strategies. In: Proceedings of the 1st International Conference on Autonomic Computing and Communication Systems, pp. 15:1–15:10 (2007)
16. Rosa, L., Rodrigues, L., Lopes, A., Hiltunen, M.A., Schlichting, R.D.: From Local Impact Functions to Global Adaptation of Service Compositions. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 593–608. Springer, Heidelberg (2009)

17. Rosa, L., Rodrigues, L., Lopes, A., Hiltunen, M.A., Schlichting, R.D.: Self-management of adaptable component-based applications. Tech. Rep. 7318, INESC-ID, Lisbon, Portugal (2011)
18. Rubio-Loyola, J., Serrat, J., Charalambides, M., Flegkas, P., Pavlou, G.: A functional solution for goal-oriented policy refinement. In: Proceedings of the 7th IEEE International Workshop on Policies for Distributed Systems and Networks, pp. 133–144. IEEE Computer Society (2006)
19. Sykes, D., Heaven, W., Magee, J., Kramer, J.: From goals to components: a combined approach to self-management. In: SEAMS 2008: Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems, pp. 1–8. ACM (2008)
20. Walsh, W.E., Tesauro, G., Kephart, J.O., Das, R.: Utility functions in autonomic systems. In: Proceedings of the First International Conference on Autonomic Computing, pp. 70–77. IEEE Computer Society (2004)

Dealing with Non-Functional Requirements for Adaptive Systems via Dynamic Software Product-Lines

Carlo Ghezzi and Amir Molzam Sharifloo

Dipartimento di Elettronica e Informazione, Politecnico di Milano,
P.zza Leonardo da Vinci 32, 20133 Milano, Italy
{ghezzi,molzam}@elet.polimi.com
http://deepse.dei.polimi.it/

Abstract. This paper focuses on the development of adaptive software, i.e., software that can automatically adapt its behavior at run-time in response to changes in the surrounding context in which it is situated. Furthermore, we focus on adaptation that is required to ensure continuous satisfaction of non-functional requirements. We propose that the implementation should be architected as a dynamic software product line (DSPL), whose target configurations can be generated dynamically. We discuss how the DSPL can be verified against non-functional requirements at design time through model checking. We also discuss how at run time the appropriate instance of the DSPL can be selected and dynamically installed and enacted as context changes are detected that can be handled correctly by such instance.

1 Introduction

Many modern software applications are embedded in an environment that can change and must satisfy requirements that also change. Changes are not under the application's control; they may occur autonomously and unpredictably. Their occurrence, on the other hand, may affect the ability of the application to accomplish its goals. At the same time, there is an increasing demand for software solutions that can recognize and tolerate changes, by evolving and dynamically adapting their behavior to provide continuous service as changes occur. This is necessary for systems that must be perpetually running and cannot be taken off-line to be updated. Because of these motivations, research on systematic development of self-adaptive systems became increasingly popular in the software engineering community during the recent years, and various research efforts are being carried out on different aspects of the development and management of these systems.

Conceptually, an adaptation is required when a violation of the requirements is detected. We are of course referring here to a broad notion of requirements, which includes not only functionality, but also quality of the delivered functionality. In other words, we refer both to *functional* and *non-functional* (or

R. de Lemos et al. (Eds.): Self-Adaptive Systems, LNCS 7475, pp. 191–213, 2013.

extra-functional) requirements [31]. In the rest of this paper, we actually implicitly restrict our focus on non-functional requirements (NFRs). We will in particular refer to NFRs that can be described quantitatively in probabilistic terms. Two notable examples, discussed in the paper, are *reliability* and *energy consumption*. While reliability concerns are traditionally considered in the case of dependable software, energy consumption has been emerging more recently due to the increasing use of battery operated devices and – more generally – energy saving concerns.

Changes in the operational environment (the *context*) are the main sources that may cause requirements violation ([16]), and hence require adaptive changes in the application software. Typical examples of environment changes are changes in the infrastructure on which the software is deployed (e.g., in the network used to interconnect the nodes of a distributed application), in the components integrated in the application and/or used as external services (e.g., in a location service that provides the current spatial coordinates of a device or in a weather forecast service that provides the weather conditions in given target geographical locations), in the usage profiles (e.g., in the case of user-intensive applications, changes in request submission rates from users may significantly affect satisfaction of the application's requirements). The phenomena that are relevant here and their changes are called *environmental* (or *contextual*) because they are not under control of the application, but rather they occur spontaneously in the environment in which the software is embedded. Although most of our discussions in the paper refer to environment changes, the approach we propose can also handle user requirements changes, provided that possible changes are anticipated during the initial design. Our approach instead does not try to take into account implementation flaws that may generate failures at run time, which also require changes in the running system. These are ignored here; that is, we assume the implementation to be correct with respect to the specification.

This paper discusses how software systems may be made self-adaptive to environment changes that may cause requirements violations at run time. The goal of self-adaptation is indeed to support dynamic configuration of a running software system so that it keeps compliance with the requirements or—at worst—in such a way that the running implementation generates minimum disruption. The proposed solution is based on a holistic approach that covers both the design and the run-time phases of the application lifetime. A design-time verification phase is integrated with continuous run-time verification and reconfiguration that support the adaptation. The need for such a holistic approach has been motivated in [2]. The solution that aims at supporting the approach is presented in this paper.

The main contributions of our work can be summarized as follows. First, at design time, the software application is designed as a *dynamic software product line* (DSPL). A software product line ([9,29]) defines a *family* of software products that can be viewed as *configurations*. In our case, configurations differ in the way they satisfy the same requirements in different contexts. The different contexts are represented during design as *variation points* in product-line terms. The product line, in our approach, is dynamic in the sense that the different instances are generated dynamically at run time. Moreover, our notion of a "product" is unconventional, in the sense that it does not refer to code, but to higher-level models. The models we refer to are the ones that are used by software engineers during design to reason about the NFRs of interest for applications. Because our main focus here is on reliability and energy consumption, we refer to Discrete Time Markov Chains (DTMCs) as our reference modelling formalisms. DTMCs, briefly summarized in Section 2, proved to be a suitable formalism to deal with our target NFRs.

All our reasoning and manipulation is performed at the model level. In particular, models are also used to describe configurations. The way models are transformed into implementations and then into deployed units is ignored in this paper. We simply assume that this can be done by following some systematic model-driven development strategy. We focus instead on how design models (in our case, DTMCs) for the various configurations can be generated from requirements. The method we propose starts from a higher level specification of the system and of the environment, given in terms of *sequence diagrams* (SDs). These provide a notation that is widely used and mastered by software developers. SDs are also extended with variants and variation points and from them it is possible to derive DTMCs in a mechanical way.

At design time, the DTMCs that describe the entire (model level) product line are verified against the requirements. Conceptually, all models of all configurations are verified to satisfy the requirements in the environment conditions for which they have been designed. Environment conditions are in fact also described as part of the models. A main contribution of this paper is to show how to avoid separate verification of each configuration through a novel approach that exploits commonalities among different configurations, which are factored out to support efficient verification, as we will discuss in Section 4. At run time, whenever the (model of the) currently running instance of the application is found to violate the requirements, because of environmental changes, another instance is identified (and the corresponding target implementation is deployed) that does satisfy the requirements under the new external conditions.

To support our run-time approach, the following framework must be in place:

1. Suitable monitors collect data that characterize the environment phenomena of interest, which may indicate potentially relevant changes.
2. The low-level data collected by monitors are abstracted into the corresponding parameters of the model M of the currently running configuration and the updated values yield a modified updated model M'. As we anticipated earlier, the models represent the behavior of both the application and the environment.
3. The updated model M' is verified against the requirements. Because we use probabilistic models, verification is performed by means of probabilistic model checking tools.
4. If requirements are violated, a new configuration M'' that satisfies the requirements is selected and the corresponding target implementation is deployed.

In previous papers [12,13] we described how continuous model update (steps (1), (2)) and verification (step (3)) can be accomplished for DTMCs. It is also important to observe that since verification is performed on the model of the application, it verifies all its possible behaviors. Therefore, it may detect a violation of requirements even for behaviors that have not been observed yet in the running system. A violation of the requirements may thus be viewed as a *prediction* that the requirements will be violated, and indicate the need for a suitable preventive reconfiguration action to be performed. This issue is discussed more thoroughly in [16]. Step (4)—which can be viewed as the heart of self-adaptation—is instead the main contribution of this paper. In addition, the paper advocates a DSPL-based design approach and shows how model checking can be applied both at design time—to verify NFR properties of the DSPL being developed—and at run time—to guide the selection of the appropriate configuration that best satisfies the requirements.

The remainder of the paper is structured as follows. Section 2 presents related work and background approaches on which this work is founded. Section 3 introduces a motivating running example. The proposed framework is then fully illustrated in Section 4. Finally, Section 5 discusses the current challenges and future work.

2 Related Work and Background Approaches

Many research efforts are presently undertaken to design self-adaptive systems that can dynamically adapt to external changes. The reader may refer to [6] and to the series of proceedings of the SEAMS workshop (now symposium) series for a comprehensive view of the different approaches being investigated. Due to the lack of space, and because of our focus, hereafter we only position our work in the context of efforts that explicitly concentrate on NFRs. We subsequently review related work in two research areas on which this paper is also rooted: dynamic software product lines and probabilistic model checking.

2.1 Self-Adaptive Systems for NFRs Satisfaction

The Rainbow framework described by Garlan et al. [7] represents one of the earliest attempts to support self-adaptation of software systems. Adaptations are prescribed as script rules for different foreseen problems at design time. Calinescu and Kwiatkowska [4] introduce a framework to implement autonomic systems in order to optimize satisfaction of NFRs. The framework mainly relies on policies specified by users, which are defined over configurable parameters. Adaptation planning is performed by exhaustively searching for optimal values of configurable parameters. Maximum size of a queue and database pool are two examples of configurable parameters. This approach does not support architectural adaptation. This can be a shortcoming because most of modern systems are comprised of black box components and only architectural adaptations are possible [26]. Adding and replacing components are two examples of architectural adaptations. Also the approach applies classical model checking against all possible configurations. This may result in inefficiencies due to large number of possible configurations.

To make adaptation planning flexible, Kim et al. [21] and Elkhodary et al. [11] exploit the use of learning algorithms. In particular, Kim et al [21] investigate the use of *reinforcement learning* techniques to enact dynamic adaptation plans at run time. They propose an approach to Q-Learning-based action planning in which in any given situation an appropriate adaptation is selected. After performing an adaptation, the system receives a reward that represents the effectiveness of the applied adaptation. The reward is used to tune the parameters of the learning functions which select the next adaptations. The main problem of the learning-based approaches is the learning period that the algorithms require for tuning parameters.

The above approaches and their characteristics are summarized in Table 1. An approach like the one pioneered by [7] uses design-time incomplete knowledge to provide adaptation rules, while others like [21] [11] are based on run-time learning. Both of them may lead to failure in adaptation planning. Prescribing adaptation plans at design time can be very risky due to incomplete knowledge. On the other hand, applying only run-time approaches needs a long learning time after system deployment. Our aim is to reach a balance between design time and run time. We embed adaptation points into system models and make sure that possible configurations can satisfy NFRs with respect to design-time assumptions. If design-time assumptions are not violated, then the running system is guaranteed to satisfy the requirements. However, in case the assumptions are violated, we apply light-weight run-time planning techniques. To be more clear, at run time we collect environmental data, update models, and apply evolutionary techniques to find candidate configurations. We ensure that a selected reconfiguration improves NFRs satisfaction. Our verification-based run-time approach is efficient because we use parametric verification techniques that are computationally expensive only at design time. Thus, the run-time overhead of planning is minimized.

Table 1. Model-based approaches to build adaptive systems

Approach	**Specification**	**Planning**	**Adaptation**	**Run-time Overhead**
Garlan et al [7]	static	design time	architectural	rule reasoning
Calinescu et al [4]	behavioral	run time	parametric	exhaustive search
Kim et al [21]	static	run time	architectural	learning

Another distinctive feature of our approach is the use of high-level behavioral models for system modelling and NFRs analysis. This is important specially due to the nature of NFRs like reliability and cost, which are highly dependent on system behaviors. For example, the number of repetitions of an activity can have an impact on reliability of a system scenario. Behavioral models can precisely predict future satisfaction of NFRs. Calinescu et al. [4] also base their approach on the use of formal models of NFRs, but they do not focus on architectural adaptations. To achieve this goal, we exploit the application of DSPLs in the design of adaptive system.

2.2 Dynamic Software Product Lines

A *software product line* SPL is defined by Software Engineering Institute as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [9]. SPL engineering (SPLE) is thus an approach to reducing the time and effort of developing a family of products. SPLE mainly focuses on developing a static product in which variation points are bound to specific variants before the application is deployed and run.

Mobile environments and modern networks demand high adaptive software systems, because their user requirements and resource constraints continuously change. These are the reasons why DSPLs become important in developing self-adaptive systems. The main difference between a SPL and a DSPL is the binding time of variation points to variants. In a SPL the binding is established *before* run time, while in a DSPL the binding is established later at run time. Indeed, while SPLs are used to deal with variability of the market, DSPLs are applied to cope with changing environments and individual requirements.

According to Cetina et al. [5], "DSPLs encompass systems that are capable of modifying their own configuration with respect to changes in their operating environment by using run-time reconfigurations". According to Hallsteinsen et al. [19] [18], DSPLs support dynamic variability, frequent run-time binding, and user requirements change. Moreover, they may provide the capabilities for context-awareness, self-adaptation, and autonomic decision making. In [23], context-awareness, resource-aware decision making, permanent service delivery, and consistent dynamic reconfiguration are considered as the major properties of DSPLs.

There exist a few research efforts aiming at modeling and developing DSPLs. Morin et al. [27] describe how DSPLs may be architected to manage dynamic adaptations. Trinidad et al. [30] model a DSPL by using feature models. Bencomo et al. [3] discuss how to capture and model variability of adaptive systems through SPL modeling approaches. They use feature models to provide a structural view of system variability, and apply transition diagrams to specify system reconfigurations in response to environmental changes. In [24,28], the authors discuss the application of aspect-oriented methods in developing DSPLs.

Lee and Kang [23] point out the importance of DSPL to address unexpected changes in environment and focus on dynamic reconfiguration as important means. They describe how to use feature models to represent variation points and how to switch between different configurations with respect to system context. Our approach follows a similar path, although our main focus is on using a DSPL-based software architecture to achieve run-time adaptations that enable continuous satisfaction of NFRs. The approach relies on efficient formal analysis of NFRs, which dynamically supports architectural adaptation planning.

2.3 Probabilistic Models for Non-Functional Requirements

Probabilistic models provide very useful and expressive power to specify uncertain and unpredictable behavior in a quantitative manner. Using randomization in distributed protocols of computer networks is one of the examples leading to unpredictable behaviors. Probabilistic model checking has been developed in the recent past to verify models that exhibit stochastic behaviors. It has been used in various domains from biological systems and chemistry to sensor networks. It can be also used for verifying and predicting non-functional properties of software systems, such as reliability, performance, and cost. The underlying models used in probabilistic model checking are different kinds of Markov models including Discrete-Time Markov Chains (DTMCs), Continuous-Time Markov Chains (CTMCs), Markov Reward models, etc. Recently, there have been great improvements on tools and techniques for probabilistic model checking. PRISM and MRMC are two important probabilistic model checkers that are currently being used [22].

In our work, to support formal analysis of requirements, we generate Markov models that are amenable to model checking. We generate DTMCs for reliability analysis and DTMCs with rewards for energy consumption. For space reasons, we only provide a sketchy introduction to DTMCs and to the property language in which properties may be expressed and analyzed. The reader may refer to [22] for details. Since in this paper we only focus on reliability and energy consumption, we ignore CTMCs, which instead are very useful to model performance.

A DTMC can be viewed as a state machine, where transitions are annotated with probabilities. For example, the DTMC in Figure 1 describes a system where in the initial states two actions may be chosen (A, with probability 0.7 and B, with probability 0.3). Execution of A may then succeed with probability 0.9, leading to state C or fail, leading to state D. Similarly, B may succeed with probability 0.99 (or fail with probability 0.01). DTMCs with reward also label transitions with real numbers representing costs (which may, for example, model the energy consumption of the activity that causes the state transition).

As for the property languages, PCTL (Probabilistic Computation Tree Logic) and cost/reward formulae have been proposed by the research community [1]. These property languages belong to the family of temporal logic languages. For example, in PCTL we can express a property concerning the reachability of a failure (or success) state. Similarly, using cost/reward formulae we can express properties on the energy consumption of a certain transaction. As an example $P => 0.98[F(State = C)]$ evaluated in state S represents the following property: "The probability of eventually reaching state C is greater than or equal to 0.98". The temporal operator F stands here for "eventually". Other examples will be given later in the paper.

Although a language like PCTL can express quite sophisticated forms of requirements, in this paper we restrict ourselves to a specific subclass (*reachability formulae*), which can describe the most common kinds of reliability requirements. For example, in this restricted language one can state that the probability that a complex transaction eventually reaches a state indicating a failure should be

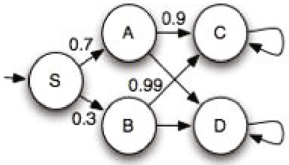


Fig. 1. A Sample DTMC

lower than a certain threshold. A similar constraint might be expressed for energy consumption, in the case where the transaction is run by a low power device. The reason for focusing just on reachability properties is to avoid delving into the details of probabilistic temporal logic. The approach we described in [13], which can be used to perform run-time verification, applies to full PCTL.

In the work we describe here, we refer to a new variation of probabilistic model checking that has been recently proposed ([10], [13]), in which probabilities of some transitions of Markov model are replaced with variables (called parameters). The resulting model is called a parametric Markov model. The probability of reaching a certain state (as expressed by a reachability property) can be computed as a rational function instead of a constant value. To evaluate the function, we need to provide real values for the parameters appearing in the function. In the rest of the paper we call these functions as parametric formulae.

The use of parametric model checking fits the application domain of adaptive software. In fact, the DTMC transitions that model changeable phenomena can be represented symbolically as parameters whose value becomes known at run time. Instead of running the classic model checker each time the values of parametric transitions change, one can much more efficiently feed values into the rational formula that describes the reachability property and perform a rather simple computation. This approach can be implemented using the PARAM model checker [17]. PARAM supports parametric model checking of parametric DTMCs and parametric DTMCs with Reward [17].

The use of formal models and model checking to verify SPLs has been pioneered by [8]. This work, however, does not deal with quantitative probabilistic requirements and does not deal with dynamic verification, two features that are instead necessary for DSPLs.

3 Running Example

In the following, a running example is described for which we aim at building an adaptive system.

"The Happy Hour Organizer (HHO) is a system to help people socialize as they move around in a modern city. The system is developed in order to make organization of daily social events easy and as automatic as possible. One of the scenarios that this system supports is about "grouping" in impromptu meetings. To achieve that, the system helps people, who have the same interests, to find each other and perform a social activity (which we call Happy Hour). For instance, someone may like to meet other people who study the same foreign language to practice in conversation. Thus they may have a nice evening in a bar while sharing their knowledge about the language. To organize such impromptu meetings, the HHO application running on the user's smartphone looks into a social network and searches for other people around city and especially near her place. The system obtains the user's current position, and takes it into account while selecting and contacting people. The application finds a group of people and communicates with their devices to make an agreement on the appointment time. Later, the application searches and books a place like a bar or pub in which the event can be held.

The system is to satisfy two NFRs concerning reliability and energy consumption. More precisely, the whole scenario shall be performed with a reliability higher that 0.95 and maximum energy consumption of 1000."

The running example described above includes variation points both in the system and environment. For example, the mobile system needs to detect the current position of the user through a *locator* device. This functionality can be performed via two embedded components: GPS or GSM. Another example is the communication service between different devices, which can be performed either via WiFi or SMS. These two are examples of internal variabilities developed as a part of system. Examples of the variation points in the environment are the *social network* and the *place booking* services. Many external applications exist to support these services, and their invocation corresponds to external variation points, whose variants can be found in the environment. Different variants may be visible or not depending on the physical location; they may appear and disappear over time; they may provide low or high-level QoS. Therefore, it is important for the system to switch between the variants which can better fit HHO's functional and non-functional requirements. Indeed, the main challenges are how to select a configuration and how to make sure that it continuously satisfies the non-functional properties. One issue is that different non-functional requirements may have a conflicting nature. For example, a reliable component may consume more energy than an unreliable one. Therefore, finding a set of variants that altogether provide a good quality with respect to the requested properties can be a difficult task.

4 The Proposed Approach

The overall view of the approach is shown in Figure 2. As illustrated, the framework covers both design time and run time. During design time, the aim is to design a DSPL, specify architectural models, and analyze them against expected NFRs. At run time, while the DSPL starts operating in environment it keeps monitoring quality data that may affect NFRs satisfaction. The requirements are continuously verified with respect to run time data that may reflect changes in the environment's behavior. In the case of detection of any violations, adaptation plans are generated and applied. In the following, we discuss each phase in turn and describe the relevant activities.

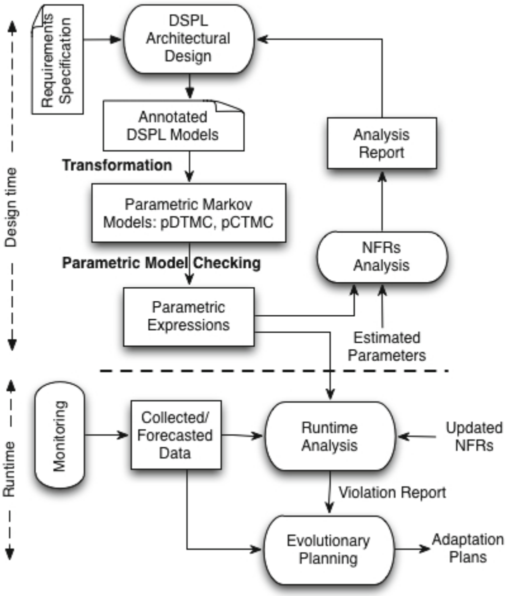


Fig. 2. The Proposed Framework

4.1 Design Time

The framework starts at design time when the architecture of the DSPL is designed through a feedback loop. The key point of design is to introduce variation points through which adaptations can be performed. The architectural design is then verified against expected NFRs by using *parametric model checking*. As we will see below, the different configurations—resulting from different instantiations of variants—are model checked in the different environment conditions for which they are conceived. The goal is to show whether or not the different configurations can satisfy NFRs. The designer can check the analysis results and may modify the architecture accordingly. In the sequel, we briefly discuss the techniques used in design-time activities.

Modeling. The main issue of modeling a DSPL is to specify variation points and variants. The system is designed as usual but the adaptive parts are specified as features, for which there exist alternative choices. The abstract feature model of the running example is shown in Figure 3. Every feature is a functionality that may be achieved using different variants. Variants can be implemented as a part of the system or may be hired as external services in environment.

In our framework, the behaviors of a system are specified by using Sequence Diagrams (SDs). Furthermore, new stereotypes (<<variation point>> and <<variable>>) are added to represent varying behaviors. The former (see Figure 4-a) describes the choice between variants in the system's architecture (*internal variability*). The variation points are represented by fragments combined through the alternative (labelled *alt* and *else*) and stereotyped as <<variation point>>. External services whose selection may be performed at run time to achieve dynamic binding are modeled as an invocation from a component, stereotyped as <<variable>>. Figure 4 shows the two kinds of varying behaviors and Figure 5 illustrates the SD for the running example. This represents an *external variability.* Concerning external services, every variation point is modeled as an invocation of a service from an abstract component, which is discovered in environment at run time. *Social Network* and *Place Booker* are two examples of external services, while *Location* is an internal variation point in the running example, referring to GPS and GSM as variants.

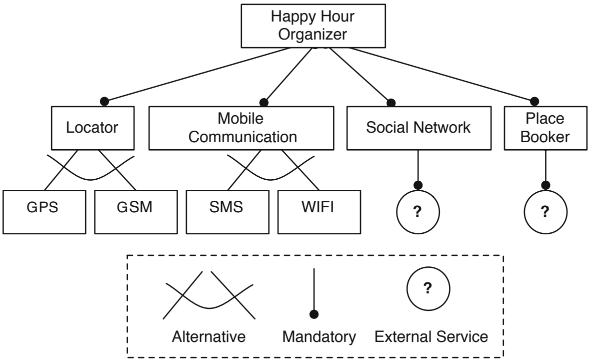
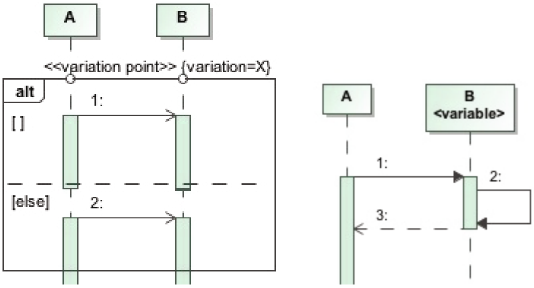


Fig. 3. The feature model for the running example



(a) Variation Point (b) Variable

Fig. 4. Varying behaviors

To evaluate NFRs, SDs can be annotated with quality data by following the UML MARTE profile. In particular, each message is annotated with two tags: *prob* and *energy.* The former represents the probability that a message is successfully transmitted; the latter expresses the amount of energy consumed to transmit a message. Figure 6 shows the annotated SD for the running example. More details of modeling and annotating SDs with quality data are described in [14].

Model-to-Model Transformation and Parametric Verification. Our goal is to ensure that non-functional requirements are satisfied by the system while it is executed. One possibility would be to use traditional model checking to achieve this goal. In this case, at design time we would model check all configurations in the different environment conditions in which they are supposed to work.

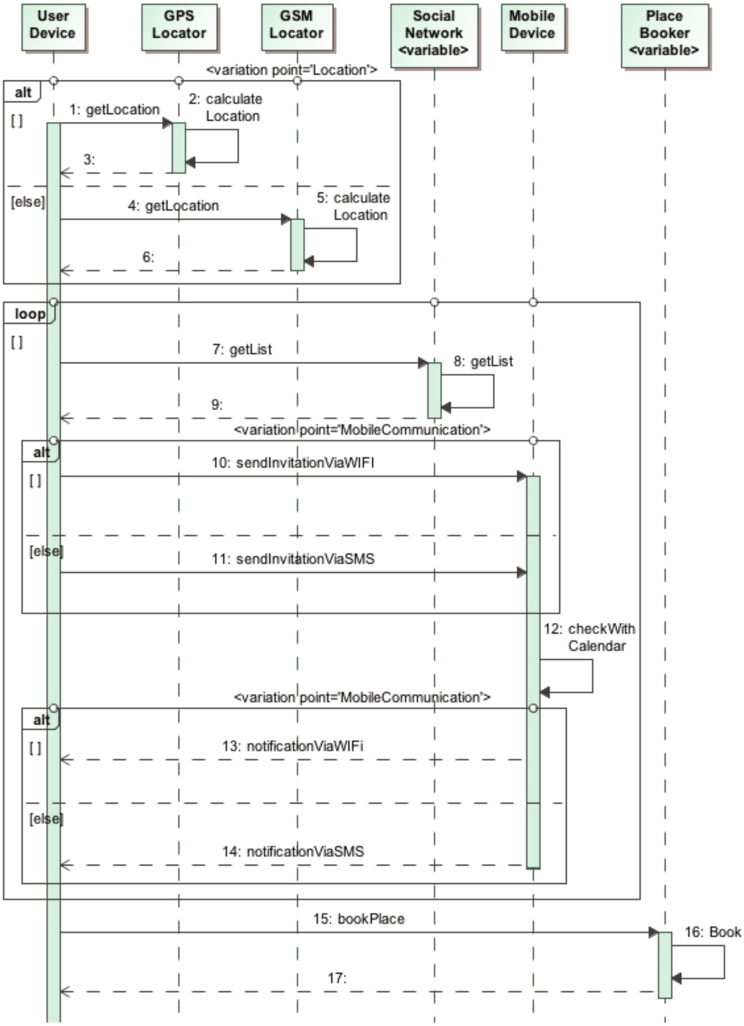


Fig. 5. The SD for the running example

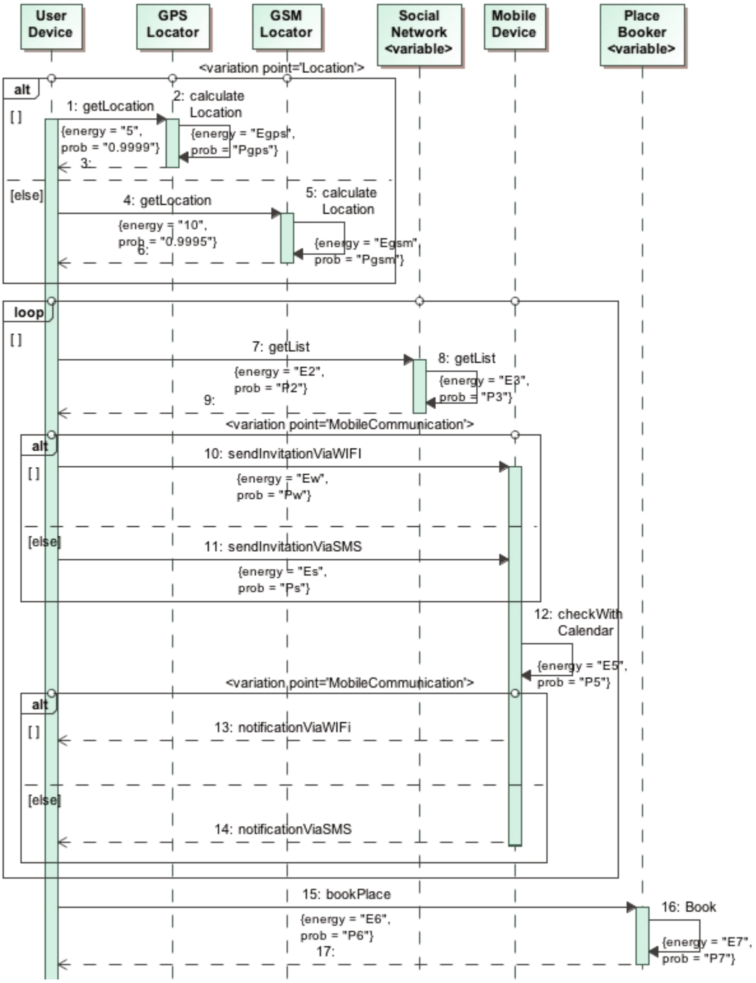

User Device
GPS Locator
GSM Locator
Social Network <variable>
Mobile Device
Place Booker <variable>
<variation point='Location'>
alt
[ ]
1: getLocation
{energy = "5", prob = "0.9999"}
2: calculate Location
{energy = "Egps", prob = "Pgps"}
3:
[else]
4: getLocation
{energy = "10", prob = "0.9995"}
5: calculate Location
{energy = "Egsm", prob = "Pgsm"}
6:
loop
[ ]
7: getList
{energy = "E2", prob = "P2"}
8: getList
{energy = "E3", prob = "P3"}
9:
<variation point='MobileCommunication'>
alt
[ ]
10: sendInvitationViaWIFI
{energy = "Ew", prob = "Pw"}
[else]
11: sendInvitationViaSMS
{energy = "Es", prob = "Ps"}
12: checkWith Calendar
{energy = "E5", prob = "P5"}
<variation point='MobileCommunication'>
alt
[ ]
13: notificationViaWIFi
[else]
14: notificationViaSMS
15: bookPlace
{energy = "E6", prob = "P6"}
16: Book
{energy = "E7", prob = "P7"}
17:

Fig. 6. The annotated SD for the running example

Whenever at run time the current configuration is executed, its model would also be analyzed by the model checker in the current environment conditions. A failure of the model checker to satisfy the requirements would then drive the selection of an alternative configuration. This approach, unfortunately, is unlikely to work in practice, especially because of the time required by the analysis step, which may lead to unacceptably late reactions. This is where parametric model checking comes into play. To make run-time verification feasible, we apply a parametric verification approach instead of the classical one. In this case, parametric verification is performed at design time and a formula is generated, which is later evaluated quite efficiently at run time when updated real data are available.

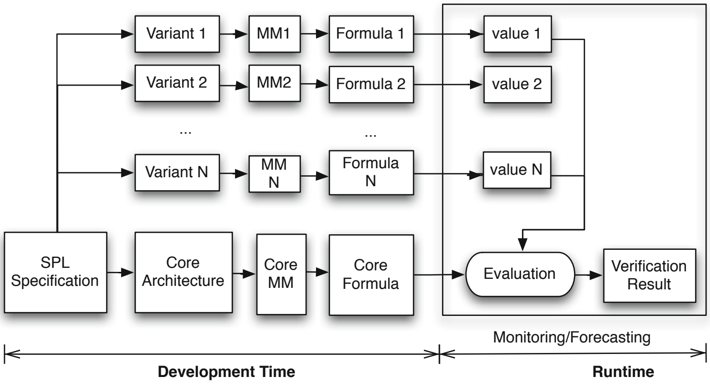


Fig. 7. Parametric Verification of SPLs - MM stands for Markov Model

Our approach is intuitively shown in Figure 7. For the sake of simplicity of presentation, we assume here that variants themselves do not contain any variation point. Note that handling nested variation points requires a simple hierarchical process, and does not impose extra effort. As illustrated, a DSPL model—an annotated SD—is divided into a core part and its variants. Through such process, variation points are transformed into messages annotated with variables. If a variation point represents an internal variability, it is transformed into a self-message annotated with two variables $P\#$ and $E\#$ standing for *prob* and *energy* respectively, meaning that their values depend on the alternative that is chosen in the configuration. The values for these variables may be computed by model checking each alternative, treated as an independent behavior. Each alternative thus goes through a similar verification process, since in general (but not in the simplified case assumed in Figure 7) it can contain further variation points and variants.

Figure 8 shows the parametric SD generated for the running example. As shown, GetLocation is an example of the self-message replaced for the variation point Location. In the case of external variation points (external services), their quality annotations are represented as variables $P\#$ and $E\#$. For instance, this applies to the place booking service of our running example. Variables are also used to label transitions that correspond to environment phenomena that may change at run time. For example, in some other interactive application we may lack information about user preferences, such as the probability that one of two alternative options may be chosen by users and may affect the way requirements may be satisfied. To model this situation, it is possible to introduce an alternative in the SD and labelling each option with a (variable) probability [1]. In conclusion, this design-time step leads to the derivation of a *parametric SD*, where variables instead of constant values are used as annotation tags.

[1] There must be an additional constraint that their sum equals 1.

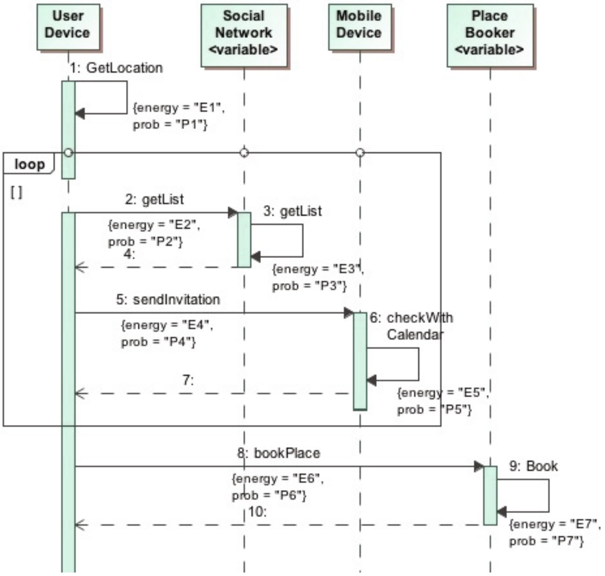

User Device
Social Network <variable>
Mobile Device
Place Booker <variable>
1: GetLocation
{energy = "E1", prob = "P1"}
loop
[]
2: getList
{energy = "E2", prob = "P2"}
3: getList
4:
{energy = "E3", prob = "P3"}
5: sendInvitation
{energy = "E4", prob = "P4"}
6: checkWith Calendar
7:
{energy = "E5", prob = "P5"}
8: bookPlace
{energy = "E6", prob = "P6"}
9: Book
10:
{energy = "E7", prob = "P7"}

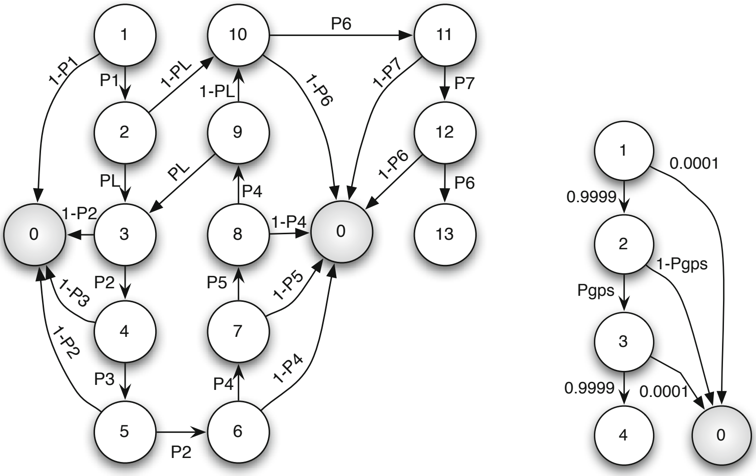
Fig. 8. The proposed framework at run time

To evaluate NFRs, parametric SDs are transformed into parametric Markov models. The transformations from SDs into Markov models are performed by following the approach described in [14], [15]. Regarding Markov models, parametric DTMCs are used to verify reliability properties, while parametric DTMCs with Rewards are used to verify cost properties (typically, energy, CPU, or network usage). NFRs are expressed as formulae written in formal languages PCTL or as Cost/Reward properties. Figure 9(a) represents the parametric DTMC corresponding to the SD in Figure 8, which can be used to reason about reliability concerns. Examples of non-functional properties we would like to state are expressed as below. Note that state 13 of the parametric DTMC (Figure 9(a)) is the state that corresponds to the condition PlaceBooked mentioned in the properties.

$$P => 0.95[F(State = PlaceBooked)] \tag{1}$$

$$R =< 1000[F(State = PlaceBooked)] \tag{2}$$

The first property states that the probability of reaching a state in which the meeting place is successfully booked shall be greater or equal to 0.95. Note that this is the final state of the whole scenario, so the property expresses a constraint on the probability of having whole scenario successfully completed. Similarly, the second property states that the whole energy consumption shall be less or equal to 1000. As mentioned earlier, both belong to the class of reachability properties.



(a) Parametric DTMC for the core behavior of the running example - state '0' is drawn two times to make the figure readable.

(b) Parametric DTMC for the alternative behavior of using GPS

Fig. 9. Parametric DTMCs

To evaluate requirements satisfaction, the parametric Markov models and the property formulae are fed into PARAM model checker. The resulting formulae of the verification for the reliability and energy properties of the running example are presented below. These formulae are used for two purposes. First they are used for design time verification of different configurations. In this case, we have to make assumptions about quality data for the parameters. The values we select represent the environment conditions we predict as possible, and for which we want to prove that an appropriate configuration exists that can satisfy the NFRs. In case no configuration is able to satisfy the NFRs, the designer should change the DSPL architecture. Furthermore, these formulae are used for run-time analysis and planning to perform continuous verification and self-adaptation.

$$Reliability = (P6^2{*}P1{*}LP{*}P7{-}P6^2{*}P1{*}P7)/(PL{*}P3{*}P2^2{*}P5{*}P4^2{-}1) \quad (3)$$

$$\begin{aligned} EnergyConsumption = {} & (2 * E6 * PL - 2 * E6 + 1 * E1 * PL - E1 - PL * E3 - \\ & 2 * PL * E2 - PL * E5 - 2 * PL * E4 + PL * E7 - E7)/(PL - 1) \end{aligned}$$

As mentioned, each variant is also transformed into parametric Markov models. Figure 9(b) shows the parametric DTMC corresponding to the selection of the GPS locator. For each variant, reachability properties are in turn evaluates on the respective Markov models. Note that in fact every variant has a behavior that starts from a starting state and ends in one or more final states. The properties are shown in the formulae below.

$$P = ?[F(State = End)] \tag{4}$$

$$R = ?[F(State = End)] \tag{5}$$

The verification of Markov models against every property also results in a formula. After providing quality data for the parameters, the formula is evaluated by substituting real numbers. The real number is the quality (reliability or energy) that a variant can provide. To evaluate the quality of the whole scenario, the quality of variants are fed into the parameters of the main formula.

4.2 Run Time

When the framework moves to run time, its activities are inspired by MAPE-K cycle shown in Figure 10, popularized by the autonomic computing research community (see [20]). The quality data collected through monitoring must be transformed into values that can be used to feed the parametric model checker. This transformation in general depends on the abstraction that model parameters realize on the physical data measurable in the environment. As a typical example, physical data may represent the detected failures of external service invocations, whereas model parameters may represent service reliability expressed as a failure probability. In general, the transformation process from environmental data collected by monitors to model parameters can be quite complex and may require approaches based on machine learning. An example is presented in [12].

Hereafter we assume that suitable transformations from monitored data to model parameters exist in the run-time environment. Updated parameters are used to evaluate the parametric formulae in order to analyze the current satisfaction and also to foresee future NFR violations.

As for the knowledge base, parametric formulae and NFR properties are kept at run time for analysis and planning purposes. In case the analysis detects or predicts any violations, planning techniques are used to generate adaptation plans by which the system can optimize its behaviors. For this purpose, we employ evolutionary algorithms and in particular Hill Climbing (HC), which is able to find a solution that represents a good trade-off between precision of the results and timeliness of the provided response. As a result, adaptation plans are generated and applied as a new configuration from the DSPL. An architectural adaptation can therefore be simply seen as set of variant substitutions for given variation points[2]. However, the main issue of planning is to find a configuration of variants that optimizes the satisfaction of possibly conflicting properties. In general, there might be various NFR properties (e.g. performance and energy consumption) that may have competing nature in the way they can be taken care of in an implementation. For example, regarding a variation point X, there may be different variants providing the same functionalities but different quality properties. Variant A may provide a high response time with a low energy

[2] The problems involved in performing the actual run-time reconfiguration are the target of another research carried out in our group [25].

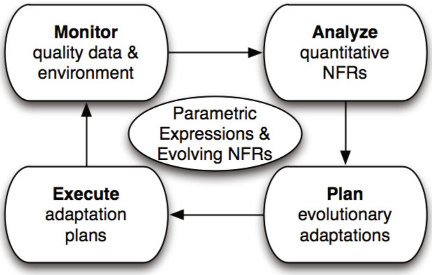


Fig. 10. The proposed framework at run time

consumption, variant B may provide a low response time and a high energy consumption, and finally variant C may provide an average quality for both cases. Due to the existence of different variation points, finding a configuration that optimally satisfies most of the NFR properties is very crucial.

If the verification of the current DSPL configuration fails at run time after updated quality parameters are fed into the verification formulae, a reconfiguration plan is activated to perform a chain of adaptations. In terms of DSPL, an adaptation is a substitution of a variant with another one that can help the system to better achieve its requirements. However, exploring all possible combinations of variants needs exponential time, and is inherently an NP-Complete search problem. It is true that by using parametric model checking and avoiding the whole run-time model checking process, the required time is reduced [15], but in case of a large number of variation points and variants, evaluating all combinations can be impossible at run time within the time limits within which a reaction to NFR violations must be enacted. Therefore, we apply an evolutionary approach, like HC, which takes into account the constraint on the available reconfiguration time and finds a sub-optimal configuration. The algorithm is able to provide a more accurate solution (i.e., one that is closer to the optimum) if more time is allocated to the search. The HC algorithm, in general, searches to find a sub-optimal solution considering a budgeted time. The search continues until a solution is reached, which represents good candidate to be chosen for adaptation, given the limited time available to perform the search.

HC is an optimization method that iteratively searches for better solutions. It starts with a random solution, then tries to improve it by iteratively changing a single element of the solution. If the change leads to a better solution, the change is applied. The process is continued until new improvements cannot be found. HC does not guarantee that the resulting solution is the best possible solution. However, it can find a better solution than other algorithms when the available search time is limited. The remainder of this section provides an intuitive, high-level description of how we apply HC to generate a new configuration of a DSPL.

Let us consider $P = \{p_1, p_2, ..., p_N\}$ be the satisfaction degree of the set of NFR properties that a DSPL is supposed to satisfy. The elements of the set represents the degree that a given configuration satisfies the properties. For each property, a weight number is introduced that expresses the importance of the property. The weights are expressed as a set of real numbers $W = \{w_1, w_2, ..., w_N\}$. Therefore, the *total utility* of a selected configuration can be specified as:

$$U_C = w_1 * p_1 + w_2 * p_2 + ... + w_N * p_N \tag{6}$$

Regarding our running example, there are two properties (p_1, p_2) corresponding to reliability and energy consumption, respectively. We consider $(w_1 = 2, w_2 = 1)$ as the weights for those properties, which means that the importance of reliability is "twice" the importance of energy consumption. Note that the energy consumption property is normalized by dividing the currently measured value by the maximum energy consumption expressed in the requirement.

Algorithm 1 shows the pseudocode for the HC approach. The algorithm starts with the current configuration of DSPL. It iteratively searches for any other configuration that can better satisfy the properties. To do that, the algorithm randomly chooses a variation point and replaces one of its variants. Then, the total utility of the new configuration is calculated. If it is greater than the utility of the current configuration, it is selected as a candidate configuration. The algorithm continues to randomly search for other candidates that are better than the new selected configuration. This procedure is carried out until the limited time is finished. In the end, the difference between the initial configuration and the selected configuration is calculated in terms of variant substitutions.

Algorithm 1. Hill Climbing Algorithm

```
$HillClimbing(VarationPoint[]VP, Variant[]VR, Configurationcf)$

$tempCf \leftarrow cf$
while $timeLimit < 0$ do
    $vnt \leftarrow ChooseVariant(VP, VR)$
    $newCf \leftarrow Combine(tempCf, vnt)$
    if $Utility(newCf) > Utility(tempCf)$ then
        $tempCf \leftarrow newCf$
    end if
end while
return Diff(tempCf, cf)
```

Let us consider as an example how the proposed approach works for our running example. Of course, the example does not show the real value of HC, which would become evident only in the case of a very high number of alternatives to evaluate. Assume that (see Figure 11) the currently running configuration for the HHO application uses GPS and SMS as the internal variants. Also assume that the user moves around and changes her physical context. It may then happen that the quality parameters change and the NFR properties (1) and (2) are not satisfied any more. The updated quality parameters are shown in Table 2. A violation is discovered by evaluating the parametric formulae (3) and (4) for both reliability and energy consumption considering the updated parameter values. In fact, the evaluation results in 0.73 for the reliability property (1), which is much less than 0.95 as the expected minimum. To deal with such violation, the HC algorithm is applied and a configuration using GSM and WiFi is selected as the new configuration (Figure 12). As the result, the application shall apply two adaptations: substituting GSM for GPS, and WiFi for SMS. Using this configuration, the reliability and energy consumption properties are evaluated to 0.95 and 836, which satisfy both NFR properties. The updated parameters of the new configuration are shown in Table 3.

Table 2. Reliability and energy parameters of the running example

Reliability	P1	P2	P3	P4	P5	P6	P7
-	0.9	0.998	0.995	0.95	0.998	0.995	0.998
Energy	E1	E2	E3	E4	E5	E6	E7
-	50	60	70	40	50	40	30

The application keeps monitoring and updating the quality parameters, and feeds them into the parametric formulae in order to discover future violations. For the sake of simplicity, we did not discuss the QoS changes of external services in this example. In the example, Facebook and BookMilano are used as the external services. Similarly to internal variabilities, it can be the case that their QoS changes which may lead to property violations and further adaptation planning.

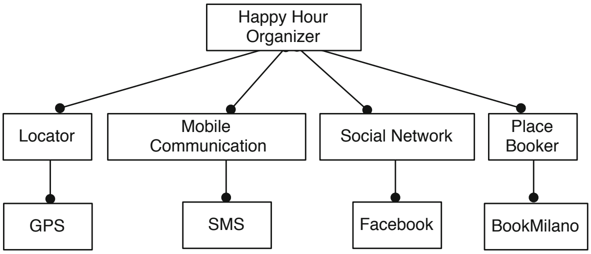


Fig. 11. The violated configuration using GPS and SMS

Table 3. New reliability and energy data for the quality parameters

Reliability	P1	P2	P3	P4	P5	P6	P7
-	0.995	0.998	0.995	0.999	0.998	0.995	0.998
Energy	E1	E2	E3	E4	E5	E6	E7
-	55	60	70	45	50	40	30

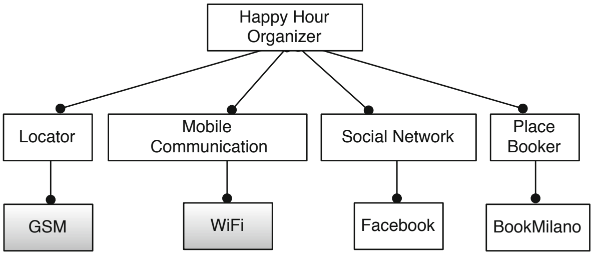


Fig. 12. The new configuration after applying the adaptations

5 Conclusion

The work described in the paper is still on going. Prototype implementations exist for various components of the development of an design environment, supporting the development of a DSPL from the requirements, and of the run-time environment, supporting dynamic verification and reconfiguration. Our future efforts will be devoted, on the one side, to developing a full-fledged environments that can support the overall framework and the seemless transition from design time to run time. On the other side, we explore other alternatives to our current approach to dynamic reconfiguration based on replanning and HC. We are currently in the process of developing an experimental evaluation of the efficiency of the HC approach on large-scale systems with many variants using simulation. We wish to explore both how the approach scales and whether alternative approaches can be devised to support continuous adaptation.

Acknowledgments. This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

References

1. Baier, C., Katoen, J.-P.: Principles of Model Checking (Representation and Mind Series). The MIT Press (2008)
2. Baresi, L., Ghezzi, C.: The disappearing boundary between development-time and run-time. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER 2010, pp. 17–22. ACM, New York (2010)
3. Bencomo, N., Sawyer, P., Blair, G.S., Grace, P.: Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In: Workshop on Dynamic Software Product Lines, pp. 23–32 (2008)
4. Calinescu, R., Kwiatkowska, M.: Using quantitative analysis to implement autonomic it systems. In: Proceedings of the 31st International Conference on Software Engineering, pp. 100–110 (2009)

5. Cetina, C., Giner, P., Fons, J., Pelechano, V.: Designing and Prototyping Dynamic Software Product Lines: Techniques and Guidelines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 331–345. Springer, Heidelberg (2010)
6. 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)
7. Cheng, S.-W., Huang, A.-C., Garlan, D., Schmerl, B.R., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer, 276–277 (2004)
8. Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., Raskin, J.-F.: Model checking lots of systems: efficient verification of temporal properties in software product lines. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010, vol. 1, pp. 335–344. ACM, New York (2010)
9. Clements, P., Northrop, L.: Software product lines: practices and patterns. Addison-Wesley Longman Publishing Co., Inc., Boston (2001)
10. Daws, C.: Symbolic and parametric model checking of discrete-time markov chains. In: International Colloquium on Theoretical Aspects of Computing, pp. 280–294 (2004)
11. 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 (2010)
12. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: Proceedings of the 31st International Conference on Software Engineering, ICSE 2009, pp. 111–121. IEEE Computer Society, Washington, DC (2009)
13. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: Proceeding of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 341–350. ACM, New York (2011)
14. Ghezzi, C., Sharifloo, A.M.: Quantitative Verification of Non-functional Requirements with Uncertainty. In: Zamojski, W., Kacprzyk, J., Mazurkiewicz, J., Sugier, J., Walkowiak, T. (eds.) Dependable Computer Systems. AISC, vol. 97, pp. 47–62. Springer, Heidelberg (2011)
15. Ghezzi, C., Sharifloo, A.M.: Verifying non-functional properties of software product lines: Towards an efficient approach using parametric model checking. In: Software Product Line Conference, pp. 170–174 (2011)
16. Ghezzi, C., Tamburrelli, G.: Reasoning on non-functional requirements for integrated services. In: IEEE International Conference on Requirements Engineering, pp. 69–78 (2009)
17. 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)
18. Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic software product lines. Computer 41, 93–95 (2008)
19. Hallsteinsen, S., Stav, E., Solberg, A., Floch, J.: Using product line techniques to build adaptive systems. In: Proceedings of the 10th International on Software Product Line Conference, pp. 141–150. IEEE Computer Society, Washington, DC (2006)
20. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36, 41–50 (2003)

21. Kim, D., Park, S.: Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In: International Workshop on Software Engineering for Adaptive and Self-Managing Systems, pp. 76–85 (2009)
22. Kwiatkowska, M., Norman, G., Parker, D.: Prism: Probabilistic model checking for performance and reliability analysis. ACM Performance Evaluation Review 36(4), 40–45 (2009)
23. Lee, J., Kang, K.C.: A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In: International Software Product Line Conference, pp. 131–140 (2006)
24. Lundesgaard, S.A., Solberg, A., Oldevik, J., France, R., Aagedal, J., Eliassen, F.: Construction and Execution of Adaptable Applications Using an Aspect-Oriented and Model Driven Approach. In: Indulska, J., Raymond, K. (eds.) DAIS 2007. LNCS, vol. 4531, pp. 76–89. Springer, Heidelberg (2007)
25. Ma, X., Baresi, L., Ghezzi, C., Panzica La Manna, V. Luy, J.: Version-consistent dynamic reconfiguration of component-based distributed systems. In: Proceedings of ESEC/FSE 2011, pp. 245–255. ACM (2011)
26. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing adaptive software. Computer 37, 56–64 (2004)
27. Morin, B., Barais, O., Jezequel, J.-M., Fleurey, F., Solberg, A.: Models@ runtime to support dynamic adaptation. Computer 42, 44–51 (2009)
28. Parra, C., Blanc, X., Cleve, A., Duchien, L.: Unifying design and runtime software adaptation using aspect models. Sci. Comput. Program. 76, 1247–1260 (2011)
29. Pohl, K., Bckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005)
30. Trinidad, P., Cortes, A.R., Pena, J., Benavides, D.: Mapping feature models onto component models to build dynamic software product lines. In: Workshop on Dynamic Software Product Lines, pp. 51–56 (2007)
31. Van Lamsweerde, A.: Requirements engineering: from system goals to UML models to software specifications. Wiley, Chichester (2009)

Uncertainty in Self-Adaptive Software Systems

Naeem Esfahani and Sam Malek

Department of Computer Science
George Mason University
{nesfaha2,smalek}@gmu.edu

Abstract. The ever-growing complexity of software systems coupled with their stringent availability requirements are challenging the manual management of software after its deployment. This has motivated the development of self-adaptive software systems. Self-adaptation endows a software system with the ability to satisfy certain objectives by automatically modifying its behavior at runtime. While many promising approaches for the construction of self-adaptive software systems have been developed, the majority of them ignore the uncertainty underlying the adaptation. This has been one of the key inhibitors to widespread adoption of self-adaption techniques in risk-averse real-world applications. Uncertainty in this setting is a vaguely understood term. In this paper, we characterize the sources of uncertainty in self-adaptive software system, and demonstrate its impact on the system's ability to satisfy its objectives. We then provide an alternative notion of optimality that explicitly incorporates the uncertainty underlying the knowledge (models) used for decision making. We discuss the state-of-the-art for dealing with uncertainty in this setting, and conclude with a set of challenges, which provide a road map for future research.

Keywords: Self-Adaptive Software Systems, Uncertainty.

1 Introduction

Self-adaptation is an effective approach in dealing with the changing dynamics of many application domains, such as mobile and pervasive systems. In response to changes in the environment or requirements, a self-adaptive software system modifies itself to satisfy certain objectives [1–3]. While the benefits of such systems are plenty, their development has shown to be more challenging than traditional software systems [2,3]. One key culprit is that self-adaptation is subject to *uncertainty* [2,3].

In general, in the field of software engineering, uncertainty is considered as a second-order concept [4]. A common misconception is that by a set of practices the effect of uncertainty can be removed to allow focusing on the "normal" behavior. Although, it is generally true that having more information decreases the amount of uncertainty [5], it is typically not possible to eliminate uncertainty altogether as it is not practical nor desirable to collect all of the information about a system. Engineering self-adaptive software is no exception. While the

R. de Lemos et al. (Eds.): Self-Adaptive Systems, LNCS 7475, pp. 214–238, 2013.

level of uncertainty could vary, it is rarely the case that a self-adaptive software system is completely free of uncertainty.

Uncertainty can be observed in every facet of adaptation, albeit at varying degrees. For instance, one reason behind uncertainty is the fact that the system's user, adaptation logic, and business logic are loosely coupled, introducing numerous sources of uncertainty [6]. Consider that users often find it difficult to accurately express their quality preferences, sensors employed for monitoring often have uncontrollable noise, analytical models used for assessing the system's quality attributes by definition make simplifying assumptions that may not hold at runtime, and so on. We refer to these factors as sources of uncertainty. All of these factors challenge the confidence with which the adaptation decisions are made. We believe considering uncertainty as a first-class concept improves the quality or sometimes even the correctness of adaptation decisions.

In spite the fact that uncertainty is prevalent in self-adaptive software systems, it is often considered in an ad hoc fashion. One reason for this is that the term *uncertainty* is a vaguely understood concept in the community, as there are many different sources for uncertainty, and not all sources of uncertainty have similar characteristics.

Some sources of uncertainty are *external*, while others are *internal*. External uncertainty arises from the environment or domain in which the software is deployed. For example, external uncertainty for a software system deployed in an unmanned vehicle may include the likelihood of certain weather conditions occurring. Software self-adaptation is one approach in dealing with the effects of external uncertainty, e.g., in a snow storm the vehicles navigator component may be replaced with a more conservative navigator to avoid a collision. On the other hand, internal uncertainty is rooted in the difficulty of determining the impact of adaptation on the systems quality objectives, e.g., determining the impact of replacing a software component on the systems responsiveness, battery usage, etc.

Moreover, not all sources of uncertainty have similar characteristics. Sometimes uncertainty is due to lack of knowledge, while other times it is due to the variation in a parameter that affects the adaptation decisions (adaptation parameter). Techniques used to mitigate one type of uncertainty may be different from techniques used to mitigate another type.

In this paper, we aim to change the status quo by first enumerating the common sources of uncertainty in self-adaptive software. We illustrate the sources of uncertainty using a robotic software system developed in our prior work. This also provides the intuition behind the challenges posed by uncertainty in this domain. We provide a more elaborate definition of uncertainty by enumerating its characteristics in the context of prior literature. To that end, we present a conceptual model for better understanding the impact of uncertainty on self-adaptive software. We also present an overview of mathematical techniques commonly used for representing uncertainty and reasoning about it.

The crux of this paper is an intuitive, yet novel, definition of what is considered to be the optimal adaptation decision under uncertainty. Realizing the same definition using fuzzy mathematical techniques in our recent work [7] has produced promising results. Finally, we provide a discussion of the state-of-the-art approaches targeted at addressing the different faces of challenge posed by uncertainty in this setting.

The rest of this paper is organized as follows: Section 2 provides an overview of a self-adaptive robotic application that is used throughout the paper for illustration purposes; Section 3 enumerates the sources of uncertainty in self-adaptive software systems; Section 4 demonstrates the impact of uncertainty on making adaptation decisions; Section 5 establishes a new definition for what is typically considered to be the optimal adaptation decision; Section 6 provides a framework for understanding uncertainty based on its characteristics; Section 7 discusses the commonly used mathematical approaches for representing and reasoning about uncertainty; Section 8 provides an overview of the state-of-the-art in this area; and finally the paper concludes in Section 9 with a summary of contributions and a set of remaining research challenges.

2 Illustrative Example

To demonstrate the ideas and help the discussion, we use a robotic software system that have been developed in our previous work [8] as a running example. The robotic software is part of a distributed search and rescue system [8] aimed at supporting the government agencies in dealing with emergency crises (e.g., fire, hurricane). Fig. 1b provides an abridged view of the robotic system's architecture. The software components comprising the robotic system range from abstractions of the physical entities, such as software controlled sensors and actuators on board the robot, to purely logical functionalities, such as image detection and navigation. Such a system may be comprised of many different execution scenarios. For instance, the bold path in Fig. 1b indicates the *Maneuver* execution scenario, which aims to safely steer the robot. The *Camera* feed is sent to *Obstacle Detector*, which runs an image processing algorithm to identify obstacles. Obstacle information is used by *Navigator* to plan the direction and speed of movement, which are then put into effect by the *Controller*.

The software components comprising this system are *customizable*, meaning that they can be configured to operate in different modes of operation. Fig. 1a shows some of the available configuration dimensions. For instance, *Power* is a configuration dimension for the *Controller* component. A *Controller* could operate in either *Energy Saving* or *Full Power* mode. A component may have many configuration dimensions.

The configuration of a software component determines its quality attributes (e.g., response time) and resource usage (e.g., memory), which could also impact the properties of the entire system. For instance, given the resource-constrained nature of the mobile robots, the configuration decisions of each component have a significant impact on the system's performance as well as its battery life. Such decisions can only be effectively made at runtime, since the system properties (e.g., available bandwidth) are often not known at design-time and may change at runtime.

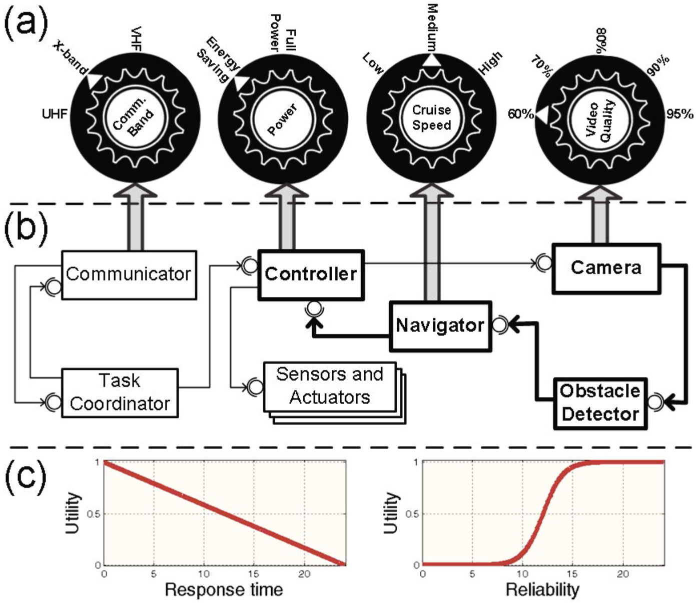


Fig. 1. A subset of the robotic software: (a) configuration dimensions and alternatives for components of the robot, (b) software architecture, and (c) utility functions defined in terms of quality attributes

As shown in Fig. 1c, for making runtime decisions, utility functions capturing the user's satisfaction with different levels of quality attribute (e.g., availability) are used. The adaptation logic uses analytical models to estimate the effect of configuration decision on the system's quality attributes, and in turn the resulting utility. For example, given the configuration of the robot's components, an analytical model, such as Queueing Network model [9], may be used to quantify the response time of a particular scenario. The objective of the self-adaptive system is to maintain a configuration for the system that achieves the maximum overall utility.

In the next section, we elaborate on the various forms of uncertainty faced by a self-adaptive software system such as this.

3 Sources of Uncertainty in Self-Adaptive Software

We borrow concepts from FORMS, a reference architecture for self-adaptive software systems developed in our prior work [10], to describe the sources of uncertainty, and exemplify them in the robotics software system. Fig. 2 depicts the high level view of a self-adaptive software system according to FORMS. In this model, the self-adaptive software system can be broken down into two parts: *Meta-Level* and *Base-Level*. The base-level subsystem provides the main functionality of the software (i.e., application logic), while the meta-level subsystem manages the base-level subsystem by reflecting on its behavior (i.e., adaptation logic). Inside the meta-level subsystem we have the MAPE-K feedback control loop [11] from IBM. In this architecture, there are four types of components that operate on the managed subsystem (i.e., base-level) and are devoted to *Monitoring*, *Analysis*, *Planning*, and *Execution* (*MAPE*). MAPE components share various models using what is known as *Knowledge* (*MAPE-K*).

The other two entities in Fig. 2 are *User* and *Environment*. The user uses the services of base-level subsystem and provides her expectations from the base-level subsystem to the meta-level subsystem by specifying objectives. For instance, Fig. 1c shows user's expectations for the robotic software system in terms of two QoS parameters (i.e., Response Time and Reliability) of the *Maneuver* execution scenario. These expectations are depicted using utility functions. The self-adaptive software system operates in an environment and hence the base-level subsystem interacts with entities from that environment. Since the meta-level subsystem is responsible for keeping the base-level subsystem on track (i.e., ensure it satisfies the user's objectives), it also needs to monitor the environment. For instance, in the robotic software system depicted in Fig. 1, the meta-level subsystem uses sensors to estimate the amount of light in the environment to adjust the *Camera* accordingly.

The entities in Fig. 2 are loosely coupled. The meta-level subsystem needs to use models of other entities in Fig. 2 as their abstractions to make adaptation decisions. The loose coupling between the meta-level subsystem and the other elements of a self-adaptive software (i.e., User, Base-Level, and Environment) is the root cause of uncertainty in self-adaptive software. Sometimes this separation among the elements of a self-adaptive software is unavoidable (e.g., distinction between system and environment), while other times it is simply necessary for enabling reuse and to manage the complexity of constructing such systems (e.g., distinction between managing parts and managed parts of a system [10,12]). We discuss the sources of uncertainty due to this loose coupling as well as a few others in the following:

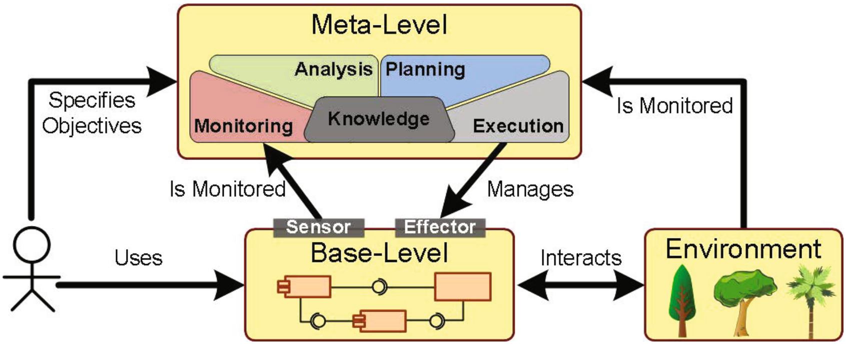


Fig. 2. High level view of self-adaptive software

- **Uncertainty due to simplifying assumptions:** This source of uncertainty is related to the *"Manages"* interface in Fig. 2 and is due to inaccuracy in the analytical models representing complex base-level subsystem. These analytical models are used to reason about the impact of adaptation choices on system's quality attributes. The error in those estimates is magnified when the modeling abstractions become inaccurate representation of the system. One of the reasons for inaccuracy is that sometimes the assumptions underlying the model are not held at runtime. For instance, an analytical model quantifying the system's response time may account for the dominant factors, such as execution time of components, and ignore others, such as the transmission delay difference between TCP and UDP. Response time estimates provisioned by such a formulation are not only error-prone, but also the magnitude of error varies depending on the circumstances. In other words, although the models are not wrong, simplifying assumptions decrease their accuracy.
- **Uncertainty due to model drift:** This source of uncertainty is related to *"Is Monitored"* and *"Manages"* interface. As we discussed earlier, for the sake of generality and reuse, the meta-level subsystem should be separated from the rest of elements in Fig. 2; therefore, due to loose coupling between the meta-level subsystem and base-level subsystem, models (knowledge) used for making decisions in the meta-level subsystem may become inaccurate representations of the base-level subsystem. Another reason for inaccuracy is the adaptation itself. Certain changes may not be enacted exactly as meta-level subsystem requests, creating a drift between the models and actual base-level subsystem. In the above example, consider the scenario in which the meta-level subsystem requests the base-level subsystem to change the communication protocol from TCP to UDP (i.e., replace a connector). If the base-level subsystem fails to enforce this change, the models used for reasoning by the meta-level subsystem become inconsistent representation of the actual base-level subsystem. Compared to the previous source of uncertainty, here we are talking about the models that over time become wrong and do not represent the base-level subsystem correctly.
- **Uncertainty due to noise:** This source of uncertainty corresponds to *"Is Monitored"* interfaces and is due to variation in a phenomenon, such as a monitored system parameter, which rarely corresponds to a single value, but rather a set of values obtained over the observation period. Consider that a sensor monitoring the available network bandwidth may return a slightly different number every time a sample is collected, even if the actual value of the bandwidth is fixed. This type of uncertainty is referred to as noise to indicate the error in the employed probes.
- **Uncertainty of parameters in future operation:** This source of uncertainty is also related to *"Is Monitored"* interfaces and is due to the actual changes in the monitored phenomenon. Without considering the behavior of the system in its future operation, a self-adaptive software may not be able to achieve its objective. For instance, our robotic software system uses sensors to measure the amount of light, which may change as the robot

navigates a terrain, to adjust the configuration of *Camera* component. The changes in light can be predicted based on the trajectory of robot movement. If the robotic software system does not consider the predictions and make decisions only based on the current amount of light, the adjustments to the *Camera* may not result in optimal improvement. Such a system is also susceptible to continuous adaptation of the system, and loss of stability, as the self-adaptation logic optimizes the system for current operating conditions, which are continuously invalidated due to changes.

- **Uncertainty due to human in the loop:** Self-adaptive software systems are increasingly permeating a variety of domains, including medical, industrial automation, and emergency response. This is partially caused by a paradigm shift from software systems used merely as data processing entities deployed on isolated servers to becoming ubiquitous and engaging the users in their daily activities. These new breeds of software often depend on correct human behavior. However, human behavior is inherently uncertain [4, 13], which in turn creates uncertainty in the software system. This type of uncertainty is related to "Uses" interface between the base-level subsystem and the user. For instance, in the case of the robotic software system depicted in Fig. 1, it is expected for the robot to interact with the rescue crew to fulfill its assignment. However, as described before, the behavior of the crew may be very unpredictable.
- **Uncertainty in the objectives:** This type of uncertainty corresponds to the *"Specifies Objectives"* interface and is due to the complexity of expressing users' requirements and eliciting preferences. While the previous source is rooted in software's dependency on human behavior, uncertainty in the objectives is the reverse relationship, i.e., it is related to human's dependency on software. In a large-scale multi-user system, users often have multiple concerns, some of which may be conflicting with one another. Eliciting user's preferences in terms of utility functions, such as those depicted in Fig. 1c, is a well-known challenge [2], as the users often have difficulty expressing their preferences and expectations using mathematical functions. Thus, the overall accuracy of such preferences remains subjective, making the analysis based on them prone to uncertainty.
- **Uncertainty due to decentralization:** In a self-organizing system several meta-level subsystems manage different base-level subsystems [3]. They create a decentralized system, where the knowledge is scattered among the self-organization units comprising the system. A self-organizing unit typically does not have complete control over the actions of other units. In such a setting, the meta-level subsystems are expected to work collectively and collaboratively to reach the system's objectives. In other words, in self-organizing software systems, the meta-level subsystem is decentralized among different entities, which makes the system prone to uncertainty. For instance, in our robotic software system, different robots may collaborate with each other to devise and update a plan for searching an area (e.g., a building that is damaged due to an earthquake) for victims with the goal of covering the area as fast as possible. This high-level collaboration adds to

uncertainty as no robot may have complete knowledge of the entire system in real-time and may not be able to control the other robots.

- **Uncertainty in the context:** Many self-adaptive software systems are intended to be used in different execution *contexts*. To that end, the meta-level subsystem is expected to detect the change in the context and adapt the base-level subsystem to behave appropriately. Portable and embedded computing devices (e.g., cell-phones) are representative of systems in this category. Here, software developers are forced to cope with additional sources of complexity introduced by the growing class of mobile and pervasive software, which are innately dynamic and unpredictable. The performance of these software systems heavily depends on availability of the resources [4], which is subject to change as the context of execution changes. For instance, in the robotic software system, a robot may move to a place in which a barrier shields its signal and prevents it from communicating with other robots, making the status of that robot unknown to the rest of system.
- **Uncertainty in cyber-physical systems:** As computation continues to become cheaper and more widespread, software and physical spaces become increasingly intertwined and tightly integrated. As a result, physical concepts are becoming increasingly important in software systems. In fact, self-adaptation capabilities are often sought after to manage the interactions between software and physical entities. This increases non-determinism and uncertainty in the software due to the fact that the physical world itself is inherently uncertain. Uncertainty caused by the effect of physical world on the software is a subset of context, which was described in the previous source. However, software can also effect the physical world, and this interaction can also host uncertainty. For instance, a robotic software system's ability to maneuver a terrain is not only a function of the accuracy of its software (e.g., routing algorithms), but also the precision in the physical steering components, as well as the physical conditions of the terrain. A self-adaptive software aimed at ensuring the robot's ability to maneuver the terrains would have to take into account the uncertainty due to the interaction between software, hardware, and physical entities in its analysis.

To mitigate uncertainty in self-adaptive software systems one should consider its sources enumerated above. Some of these sources (e.g., cyber-physical systems) have been observed in other fields of science and there are well-established approaches for addressing them. On the other hand, some of these sources (e.g., model drift) are relatively new and specific to self-adaptive systems, hence new approaches may need to be devised for addressing them.

4 Impact of Uncertainty on Self-Adaptive Software

Uncertainty has a significant impact on a self-adaptive software system's ability to satisfy its objectives. Prior research for the most part have ignored the challenges posed by uncertainty, which hamper their adoption in real-world risk-averse domains. We collectively refer to these as the *traditional approaches*. We

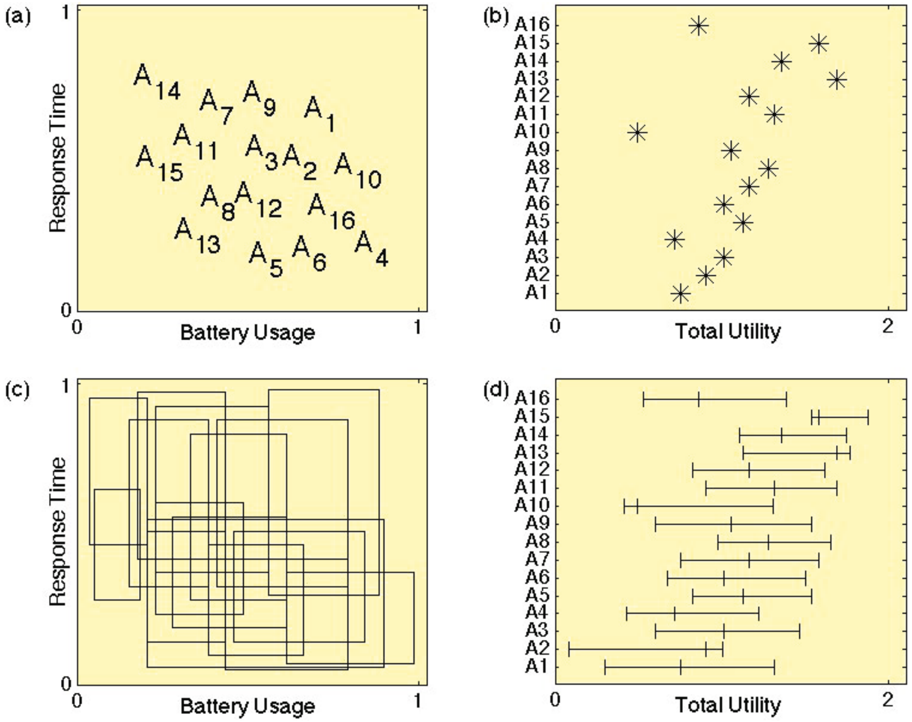


Fig. 3. Impact of uncertainty on the process of making adaptation decisions to satisfy the system's objectives: (a) 16 candidate configurations in a battery usage and response time trade-off, (b) application of utility function to resolve the trade-offs, (c) battery usage versus response time under uncertainty, where each rectangle represents the space of values that an architecture may take, and (d) the range of utility values expected for the 16 configurations under uncertainty

illustrate their shortcoming using an instance of the robotic software system in which the objective is to choose from a pool of 16 candidate configurations, such that *battery usage* and *response time* are minimized.

The traditional approaches assume that the impact of candidate configurations on properties of interest can be precisely estimated. If that was the case, then one could visualize the situation as in Fig. 3a. Here, for the sake of clarity, the values for *response time* and *battery usage* are normalized between zero and one. Assuming both properties have the same level of importance, to compare the 16 configurations, for each configuration we first sum up the values obtained from the corresponding utility function. Recall that utility functions are used to quantify the users' preferences with the values attained in properties. Fig. 3b achieves just that, as it shows the overall value for the candidate configurations. In this space, configurations can be compared with one another. For example, we can see that A_{13} is the best configuration, as it obtains the largest total value.

While the aforementioned approach is theoretically sound, it is not useful in practice, as it does not incorporate the underlying uncertainty in every facet of the approach, including the fact that analytical models often cannot precisely quantify the impact of alternative configuration on properties of interest (i.e., there is always some amount of noise), the utility functions may not be accurately representing the users' preferences, etc.

The complexity of incorporating uncertainty in the analysis is shown in Fig. 3c. Here, the uncertainty is represented in terms of range of impact that a configuration candidate may have on the properties of interest. For example, the impact of a given configuration on battery usage is no longer a single number, but rather a range of values. As a result, each configuration candidate may obtain a value anywhere within the area occupied by the corresponding rectangle. Clearly, comparing two configurations with overlapping rectangles is difficult.

The rectangles in Fig. 3c can be transformed to a space where the trade-off analysis can be performed by applying the utility function on the most optimistic and pessimistic behavior of a given configuration. Fig. 3d shows the resulting range of behavior that one would expect, assuming that uncertainty in various facets of the system can be quantified. Unlike the earlier example, it is not clear what is the optimal configuration, as the behavior of each configuration is now specified as a range, and the ranges offer trade-offs. As described in the next section, there is a need for an alternative definition of optimality in this setting that explicitly takes the uncertainty into consideration.

To gain a better appreciation for the complexity of this problem consider that the simple example used in Fig. 3 consists of only 16 configuration candidates and 2 properties of interest, but a typical self-adaptive software system often consists of many more candidates and properties. Manually exploring and solving this problem is a big burden. Incorporating uncertainty into the analysis makes a problem that is already challenging, so overwhelmingly complex that a manual assessment without the appropriate tools and techniques becomes impossible, which has been the motivation for this research.

5 Reconceptualizing Optimality under Uncertainty

We argue that to tackle the complexity introduced by uncertainty, we need to reconceptualize the definition of the optimality in self-adaptation decision making process to account for the uncertainty underlying the analysis. We provide an intuitive overview of a new definition of optimality and use the robotic software example from the previous section to illustrate it.

Figure 4a shows the shortcomings of the prevalent definition of optimality in making adaptation decisions while ignoring uncertainty. The system is initially executing with utility U_1 prior to time T_1. At time T_1, due to either an internal or external change, the systems utility drops to U_2. By time T_2, the self-adaptation logic detects this drop in utility, finds and effects an optimal configuration, which is conventionally defined as the one achieving the maximum utility. As shown in Fig. 4a, this corresponds to U_3, which represents the expected utility of the

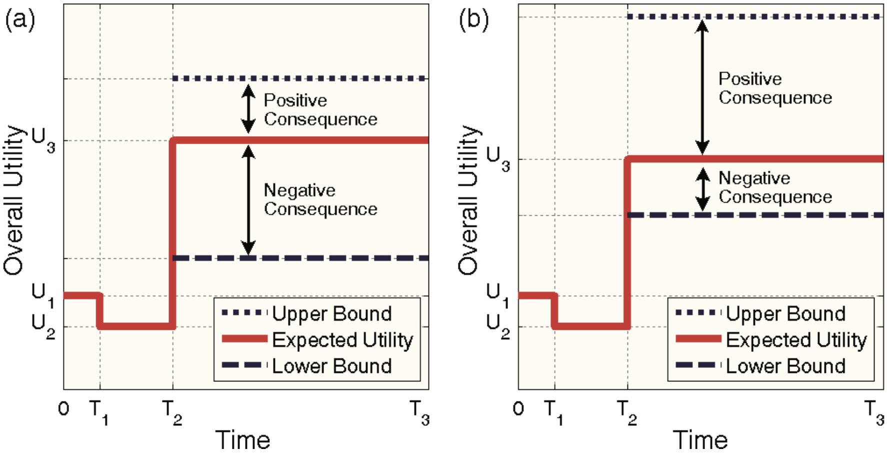


Fig. 4. The utility of a self-adaptive system based on the decision using: (a) traditional definition of optimality, where the uncertainty is not considered, and (b) advocated approach, which considers uncertainty

best configuration for the system. In practice, however, the actual utility of the system may vary between the two dashed lines, representing the likely positive and negative consequences of uncertainty. By not accounting for uncertainty, the approach is vulnerable to gross overestimation of the utility. In other words, the selected optimal solution is rather risky, and in the worst case may be a very poor choice.

We propose an alternative definition of optimality in making adaptation decisions that incorporates uncertainty. Similar to the scenario of Figure 4a, a new configuration is effected at time T_2, except we say a configuration is optimal if it concurrently satisfies the following three objectives: (1) maximizes U_3, which represents the most likely utility for the system under uncertainty; (2) maximizes the *positive consequence of uncertainty*, which represents the likelihood of the solution being better than U_3; and (3) minimizes the *negative consequence of uncertainty*, which represents the likelihood of the solution being worse than U_3.

The new concept of optimality defined above can be realized using several alternative mathematical approaches (e.g., both probabilistic and fuzzy numbers could be used to indicate the extent of uncertainty). Regardless of how the optimality criteria is realized, we can make a general observation. As depicted in Figure 4, concurrent satisfaction of the three objectives may result in a smaller value of *expected* utility (i.e., U_3) using this approach compared to that of the traditional approach. But since the information used to estimate the *expected* utility is uncertain, expected utility is not guaranteed to occur in practice. Therefore, it is reasonable to argue that the true quality of a solution is determined by the range of possible utility.

Furthermore, we argue that the new notion of optimality could be extended to also account for uncertainty in the future operation of a software system.

Figure 5a depicts a configuration picked by the traditional approach in which uncertainty in future operation of the system is neglected. As a result, a solution with the highest utility may actually be a very bad choice, since due to uncertainty in future operation of the system, it may in effect obtain a very low utility. Note that for illustration in Figure 5 the behavior over time is depicted linearly, but in general the behavior over time may follow a different trajectory.

Given the variability in system and environmental parameters, an optimal solution is not the one that achieves the highest utility at the point in time in which the decision is made, but the one that anticipates the future behavior (potentially in the form of a probabilistic prediction such as the ones obtained from Hidden Markov Models [14]) of the selected configuration over time. As depicted in Figure 5b, the optimal solution is the one that considers the behavior of the selected configuration over time, i.e., selects a configuration that may have a lower utility at the moment in which the decision is made with the expectation of achieving a better utility over a period of time in future. Another benefit of the new optimality criteria advocated here, but not depicted in the figure, is that since under the reconceptualized notion of optimality the system is expected to maintain a higher utility in its future operation, our approach decreases the number of adaptations compared to the traditional approach. This in essence results in more stable self-adaptive software systems.

These two extension (i.e., Figures 4 and 5) can also be combined. As a result, the range will be formed around the trend line and the size of the range can vary for different points in time.

We believe this new model of reasoning about optimality provides a good foundation for studying the role of uncertainty in self-adaptive software. In our recent work [7] we have used fuzzy mathematical techniques to realize the new

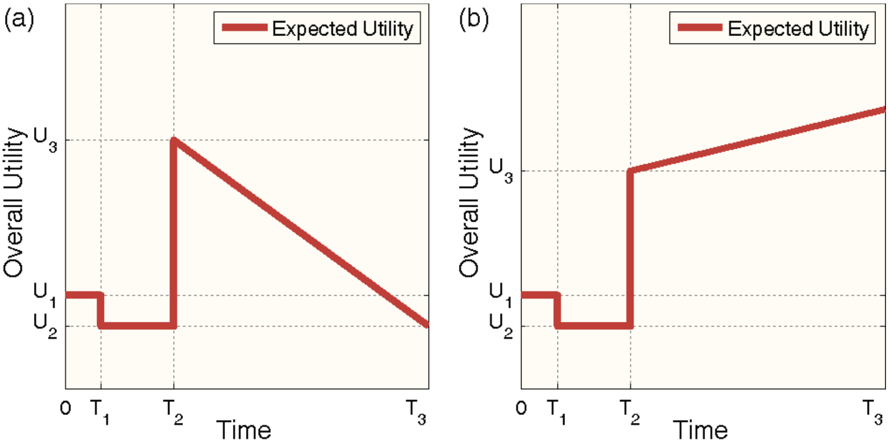


Fig. 5. The utility of a self-adaptive system over time: (a) traditional approach, where the behavior over time is not considered in the selection of a configuration, and (b) advocated approach, which considers the behavior a system in a given configuration over time

model of optimality, which has produced promising results. Our experience shows that the revised definition of optimality increases the accuracy of adaptation decisions, and allows for construction of self-adaptive software that is resilient to fluctuations in the system properties and environmental parameters. While our experience with realizing this approach using fuzzy mathematics has been promising, we believe there are other methods of realizing the approach outlined above (e.g., Bayesian probabilities), as further detailed in Section 7. Finally, as we describe in Section 8, some researchers have already observed the limitation of the existing definition of optimality (i.e., traditional approach) and have investigated possible solutions to this limitation.

6 Uncertainty Distilled

All sources of uncertainty in self-adaptive software do not have the same characteristics. Although there are some philosophical debates about the true distinction between the different types of uncertainty (e.g., [15]), it is commonly agreed that it is useful to categorize different types of uncertainty in practice. This is because the approaches for modeling different kinds of uncertainty are very different from one another. For instance, often times it is not possible to represent the user's uncertainty in the specification of objectives in terms of utility functions as a probability distribution, since the uncertainty is due to the lack of knowledge, and not variability. In the following subsections we enumerate the different characteristics of uncertainty, which we believe in turn sheds light on the appropriate techniques that should be used to tackle the different sources of uncertainty.

6.1 Reducibility versus Irreducibility

When something is inherently unknowable, the uncertainty associated with it is irreducible. On the other hand, the uncertainty associated with knowable things which are unknowns at a given time is reducible. Sometimes distinction between these two kinds of uncertainty becomes a philosophical problem, which depends on the point of view. One of the main reasons behind irreducible uncertainty is intractable complexity of phenomena with existing progress in science. For instance, it is a known fact that the physical world behaves in a non-linear fashion; however, there is little known about non-linear mathematics. Instead, non-linear phenomena are modeled using linear mathematics and hence the models have irreducible uncertainty. One may argue that this kind of uncertainty is not inherently irreducible as it can be mitigated by studying non-linear mathematics. In this paper, we stay away from philosophical debates as we want to study the practical aspects of uncertainty.

6.2 Variability versus Lack of Knowledge

From a different perspective uncertainty can be categorized as aleatory or epistemic [5]. The root of aleatory is the Latin word ãleãtor, which means gambler, while the root of epistemic is the Greek word epistemé, which means scientific knowledge. Aleatory uncertainty captures the uncertainty that is caused by randomness and is usually modeled using probabilities. On the other hand, epistemic uncertainty corresponds to lack of knowledge and sometimes is referred to as parameter uncertainty. *This distinction is motivated by the location of the uncertainty — in the decision-maker or in the physical system.* [5] In other words, variability is considered as uncertainty in the studied system, while lack of knowledge is considered as uncertainty on the decision-maker's side.

It may be tempting to map variability to irreducibility and lack of knowledge to reducibility. However, this is not generally true. For instance, if irreducible uncertainty directly implies variability, the next recipient of Turing Award, which in not known right now, would be a random phenomenon! Similar to the philosophical argument about reducibility versus irreducibility, there are arguments about distinction between aleatory and epistemic uncertainties. For instance, some argue that variability observed in the world is due to limitation of scientific models and hence lack of knowledge [15]. While these arguments are true, we should mention that these distinctions are relative and depend on the point of view. In other words, it is true that sometimes a phenomenon, which is uncertain due to variability from a given point of view, can be uncertain due to lack of knowledge from a different point of view, but, this does not mean that variability is not a characteristic of uncertainty.

Both the reducible and irreducible uncertainties can have aleatory and epistemic components. Aleatory and epistemic represent the essence of uncertainty, while irreducible and reducible represent the managerial aspect of uncertainty.

6.3 Spectrum of Uncertainty

Fig. 6 depicts the spectrum of uncertainty. *Current Information* falls anywhere between *Ignorance* and *Certainty*. The range between the *Current Information* and *Certainty* is the *Imprecision*. *Complete Information* indicates the threshold where all the knowable are known and falls anywhere between the *Current Information* and *Certainty* (i.e., inside *Imprecision*). In a sense, the *Complete Information* is a limit for the *Current Information* indicating the maximum amount that the uncertainty can be reduced. Therefore, the range between the

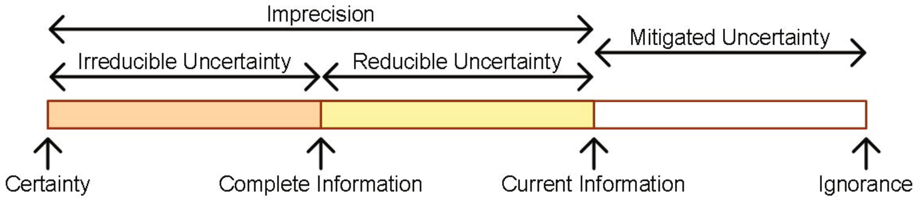


Fig. 6. The spectrum of uncertainty based on the knowledge (adopted and extended from [5])

Current Information and the *Complete Information* is the *Reducible Uncertainty.* On the other hand, the range between the *Complete Information* and *Certainty* indicates the *Irreducible Uncertainty.*

Based on the nature of a given system, the length of any of these ranges (i.e., imprecision, reducible uncertainty, and irreducible uncertainty) can be zero. For instance, when the complete information and certainty point to the same spot, there is no irreducible uncertainty. This definition also implies the fact that, as the current information increases and approaches the complete information, the imprecision becomes mainly due to irreducible uncertainty. Usually as the current information gets closer to the complete information, increasing the knowledge becomes more expensive. Sometimes increasing the knowledge may not even worth spending resources, as the added value becomes limited. We revisit this issue in the next section.

6.4 Characterizing the Sources of Uncertainty

Table 1 characterizes the sources of uncertainty based in relation to the spectrum of uncertainty. To that end, we specify if a source of uncertainty is due to variability or lack of knowledge.

Uncertainty related to *Simplifying assumptions*, *Drift*, *Human in the loop*, *Objectives*, *Decentralization*, and *Cyber-physical systems* are due to the lack of knowledge. Be it for the complexity of the models, loose coupling, ambiguity, or distribution, the lack of complete knowledge in these facets of self-adaptation makes the adaptation decisions prone to uncertainty.

On the other hand, uncertainty related to *Noise*, *Parameters over time*, and *Context* is due to the variability. In this case, uncertainty is rooted in the fact that the behavior of the system may change after the adaptation decision is made.

We drew the conclusions presented in Table 1 from examples of the sources of uncertainty that we have found in the literature, as well as our own prior experiences with the construction of such systems. Some of these examples were enumerated in Section 3. Since it is possible to have several sources of uncertainty in a single phenomenon, uncertainty related to that phenomenon may be

Table 1. Characteristics of different sources of uncertainty

	Simplifying assumptions	**Model drift**	**Noise**	**Parameters over time**	**Human in the loop**	**Objectives**	**Decentralization**	**Context**	**Cyber-physical systems**
Variability			✓	✓				✓	
Lack of Knowledge	✓	✓			✓	✓	✓		✓

both due to variability and lack of knowledge. For instance, one may make a *Simplifying assumptions* and approximate the *Noise* of a given parameter by a well-known probability distribution even if the value of that parameter does not exactly follow the distribution.

7 Mathematical Techniques for Representing and Incorporating Uncertainty

This section provides an overview of two widely applicable approaches for representing and incorporating uncertainty in self-adaptation. As will be described in the next section, existing state-of-the-art has often relied on one of these approaches.

7.1 Probability Theory

Probability theory [16] is the most widely used approach to represent uncertainty. Humans have long observed that some events are to some extent predictable. Mathematical probabilities, which are dated from 18th century, were an approach to study the regularities in the games of chance. Nowadays, probability is learned mainly through Kolmogorov's axioms [17], which allows for adoption of probability theory in broader class of problems (e.g., physical, social, industrial, etc.). Most researchers are familiar with the mathematics of probability but quite few are aware of philosophical debates regarding different interpretations of probability. Therefore, here we focus on interpretations of probability. The prominent interpretations of probability until late twentieth century were classical and frequentist interpretations.

Probability theory was originally conceived with the classical interpretation. As we mentioned, probability was originally rooted in the games of chance, and so was the classical interpretation. A fundamental assumption in classical probability is the fact that all the outcomes of a phenomenon are equally probable. This assumption is shown to cause inconsistencies when it is used in more general problems (i.e., beyond games of chance). Motivated by the limitations of the classical interpretation, the frequentist interpretation was developed. In this interpretation the probability of an event is defined as limit of its relative frequency in large number of trials, hence the name of this interpretation is frequentist. Although this definition goes beyond classical definition, it narrows the scope of the frequentist interpretation to repeatable, random phenomena.

Bayesian theory [18] is based on subjective interpretation of the probability. In this interpretation the probability is defined as an expression of a rational agent's degrees of belief about uncertain propositions. The scope of this interpretation is more general than frequentist interpretation as it extends the definition of probability by allowing probability assignment to a single experiment regardless of whether it is part of a larger number of experiments or not. Therefore, Bayesian could be used in the problems in which there is not enough data for frequentist interpretation. For instance, frequentists cannot analyze a new disease for which enough data is not available, while Bayesians can use subjective information based on related diseases to analyze the new disease.

Bayesian inference is as old as probability. However, it was disfavored due to positive orientation of Western nineteenth and twentieth century science, which was considering subjectivity to be non-scientific. Moreover, complex Bayesian models require large amount of computation, which were not possible until late twentieth century. With computational advances in the late twentieth century there has been a resurgence towards Bayesian approaches as they are a unified theory for both data-rich and data-poor problems. Many modern machine learning methods are based on Bayesian principles.

7.2 Fuzzy Sets and Possibility Theory

Fuzzy set theory [19] is an extension of classical set theory. In classical set theory, the membership of an element in a set is a binary condition: the element is either in the set with membership value of 1 or it is not in the set with the membership value of 0. However, in fuzzy set theory, the membership of an element in a set is not a binary condition, but rather a "sort of" concept. To that end, the membership value of an element with regard to a set is any value between 0 and 1. The higher the membership value is, the more likely that element belongs to the set. Therefore, the boundary of a fuzzy set is not clearly defined, whereas the boundary of a classical set is *crisply* defined.

Fuzzy sets can be applied to domains where the information is incomplete or imprecise. For instance, fuzzy sets have been used in linguistics to deal with vagueness and ambiguity of the statements. For instance, temperatures that are considered to be cold and warm are not uniquely defined and they may be different from person to person. In fact, there are some temperatures that can be considered both cold and warm to some extent. A program that tries to understand written text can use the fuzzy definition of coldness and warmness to have a better understanding of the text.

Possibility theory [20] is a theory for handling incomplete information, which is based on fuzzy sets. Among several interpretations of possibility theory, the basic interpretation is the most common one. This interpretation defines possibility as a mapping from the power set of sample space to any value between 0 and 1. In other word, any event, which is a subset of sample space, has a possibility defined by this mapping. One of the reasons that fuzzy logic is adopted in engineering is the simplicity and efficiency of its operations.

While probability theory deals with the statistical characteristic of data, possibility theory focuses on the meaning of data. There are several studies [21, 22] about the relationships of the two theories. Although, sometimes the two theories can be used interchangeably, it has been shown that the two theories are different. Some researchers have described the usability of two theories using an spectrum: possibility theory is useful when there is little information, however, when more information becomes available it is better to use probability theory.

8 State-of-the-Art

The research community has made great strides in tackling the complexity of constructing self-adaptive software systems [1–3]. However, as corroborated by others [2, 3], there is a dearth of applicable techniques for handling uncertainty in this setting. A few researchers have recently begun to address uncertainty. Table 2 summarizes their work with regard to the sources of uncertainty they are dealing with. In the following subsections we provide an overview of these approaches.

8.1 Rainbow

Cheng and Garlan [6] described three specific sources of uncertainty in self-adaptation (problem-state identification, strategy selection, and strategy outcome) and provided high-level guidelines for mitigating them in Rainbow framework [12]. Problem-state identification is related to Monitoring and Analysis activities from the MAPE loop, while strategy selection and strategy outcome are related to Planning and Execution activities, respectively. In other words, they try to mitigate uncertainty in the activities of the adaptation feedback control loop.

To mitigate uncertainty in problem-state identification, they use running average in monitoring to counter variability and stochastic properties of the environment. The observations are then compared with architectural descriptions that are augmented with probabilistic information to detect trend of behavior. Once the problem is detected, a strategy is selected to resolve the problem. The uncertainty in strategy selection is mitigated by using the *Stitch* language. This language allows for modeling uncertainty in strategies. Therefore, when Rainbow attempts to select a strategy at runtime, it can decide based on the expected value (which is capturing the uncertainty) of different strategies. Finally, once a strategy is selected and put into effect, it may succeed or fail. Instead of dealing with this uncertainty in the next adaptation loop, they consider the uncertainty in strategy outcome by specifying how long Rainbow should monitor the implementation of the strategy before committing to the change. This is another attribute of the approach that can be modeled using the Stitch language.

By augmenting architectural models with probabilistic models, Rainbow mitigates the uncertainty due to simplifying assumptions and noise. Moreover, by monitoring the system after adaptation Rainbow mitigates the uncertainty due to drift in the architectural models.

8.2 RELAX

Whittle et al. introduced RELAX [23], a formal requirements specification language that relies on Fuzzy Branching Temporal Logic to specify the uncertain requirements in self-adaptive systems (i.e., as indicated in Table 2, RELAX uses possibility theory to deal with the uncertainty of the *Objectives*). RELAX allows for explicit expression of environmental uncertainty and its effect on requirements. Depending on the state of environment, RELAX specifies the requirements that can be disabled or "relaxed". To that end, RELAX introduces a set of operators that can be used in forming the requirements. These operators also define how the requirement can be relaxed at runtime. Moreover, the operators capture the kind of uncertainty (*uncertainty factor*) that can initiate the relaxation of requirements.

In a subsequent publication [24], Cheng et al. extended RELAX with goal modeling to specify the uncertainty in the objectives. They first build the goal lattice and then use it in a bottom-up fashion to look for sources of uncertainty, which are the elements of domain/environment and can endanger satisfaction of goals. In their approach, they identify uncertainty through a variation of threat modeling, which is used to identify security threats in a system. Once the uncertainty is identified, its impact is assessed to devise mitigation tactics. The ultimate tactic for mitigating uncertainty (when all other tactics fail) is to add flexibility to the goal by "relaxing" it.

8.3 FLAGS

FLAGS [25] also uses possibility theory to mitigate the uncertainty of the *Objectives*. Similar to RELAX, FLAGS aims to achieve the basic goal of adaptive systems at the requirements level: mitigate the uncertainty associated with the environment and new business needs by embedding adaptability in the software system as early as requirement elicitation. In other words, FLAGS considers self-adaptation as a special kind of requirement, which affects other requirements. These special requirements are called adaptive goals and FLAGS allows for the

Table 2. The mathematical theories that are used by existing approaches for dealing with sources of uncertainty

	Simplifying assumptions	**Model drift**	**Noise**	**Parameters over time**	**Human in the loop**	**Objectives**	**Decentralization**	**Context**	**Cyber-physical systems**
Rainbow	Prob.		Prob.						
RELAX						Poss.			
FLAGS						Poss.			
FUSION	Prob.	Prob.						Prob.	
ADC				Prob.					
RESIST	Prob.	Prob.	Prob.	Prob.				Prob.	
POISED	Poss.		Prob.			Poss.			

definition of counter measures that must be performed if some goals are not fulfilled as expected (due to predicted uncertainty).

FLAGS also deals with another source of uncertainty in addition to the uncertainty in the context of the software: the uncertainty in the goals themselves. As satisfaction of some goals cannot be specified by simple yes–no answer, FLAGS relies on fuzzy goals for which properties are not fully known, the complete specification is not available, and small temporary violations are tolerated. Therefore, FLAGS ends up with two sets of goals: crisp goals and fuzzy goals. It formalizes the crisp goals using Linear Temporal Logic (LTL), and fuzzy goals using fuzzy temporal language, which in the end is unified with the LTL specification. Therefore, all the software requirements can be specified in a single coherent language.

8.4 FUSION

FUSION [26] is a learning based approach to engineering self-adaptive systems. Instead of relying on static analytical models that are subject to simplifying assumptions, FUSION uses machine learning, namely Model Trees Learning (MTL) to self-tune the adaptive behavior of the system to unanticipated changes. This allows FUSION to mitigate the uncertainty associated with the change in the context of software system as it gradually learns the right adaptation behavior in the new environment. The result of learning is a set of relationships between the adaptation actions in the system and the quality attributes of interest (e.g., response time, availability). These rules consider the interaction of adaptation actions and hence to some extent mitigate the uncertainty caused due to synergy. The quality attributes of interest could be measured and collected from the running system through instrumentation of the software or sensors provided by the implementation platform. The adaptation actions correspond to variation points in the software that could be exercised at runtime.

FUSION has two complementary cycles: learning cycle and adaptation cycle. The learning cycle relates the measurements of quality attributes to the adaptation actions. The learning cycle constantly monitors the environment to find possible errors in the learned relations. Persistence of such errors, which can be either due to drift or change in the context, triggers relearning the new behavior. When quality of software decreases over time and drops below a certain threshold, the adaptation cycle kicks in and uses the learned knowledge to make informed adaptation decision to improve the quality attributes. The quality of the software system is defined as aggregate collection of individual quality attributes. However, since some quality attributes may conflict with each other, the notion of utility is used to allow for making trade-offs.

8.5 Anticipatory Dynamic Configuration (ADC)

Poladian et al. [27] studied dynamic configuration of resource-aware services, where they showed how to select an appropriate set of services to carry out a user task, and allocate resources among those services at runtime. The original work did not consider the uncertainty in the environment. Subsequently, the work was extended to make anticipatory decisions [28], and considered the inaccuracy of future resource usage predictions. To that end, they built on the previous work of one of the authors [29] and used historical profiling to find an application's resource requirements for different configurations. Considering resource availability over time mitigates the uncertainty in monitoring as it provides more accurate models of the environment being monitored. As indicated in Table 2, they use probability theory to achieve this (i.e., Mitigate the uncertainty related to *Parameters over time*).

By considering the resource availability prediction, the anticipatory model of configuration chooses a configuration that maximizes the cumulative expected value of utility over time. This reduces the number of possible future reconfigurations and as a result disruptions in the system. In making the adaptation decisions, the cost of switching between the configurations is also considered. If the cost of switching is low, this approach selects a configuration that performs better at the moment and when the quality of selected configuration drops the configuration is switched. On the other hand, if the cost of switching is high, *a temporal under-optimum configuration* is accepted. That is, from the beginning an alternative configuration, which performs better over time, is selected to prevent switching later on.

8.6 RESIST

RESIST [14] uses information from several sources, such as monitoring internal and external software properties, changes in the structure of the software, and contextual properties to continuously furnish refined reliability predictions at runtime. The up-to-date reliability predictions express the reliability of the system in near future using probabilities. These predictions are then used to decide about changing the configuration of the software to improve its reliability in a proactive fashion. RESIST is targeted for *situated software systems*, which are prominently mobile, embedded, and pervasive. The uncertainty in these systems are prevalent as they have highly dynamic configuration, unknown operational profile/context, and fluctuating conditions, yet they are usually deployed in mission critical environments (e.g., emergency response) and have stringent reliability requirements. RESIST mitigates the uncertainty due to the context and simplifying assumptions through constant learning. Moreover, slight changes in the reliability are modeled as probability distributions indicating the noise.

RESIST takes a compositional approach to reliability estimation; the process starts with analysis at the component level, which in turn makes it possible to assess the impact of the adaptation choices on the system's reliability. The component level reliability is estimated stochastically using a Discrete Time Markov Chain and in terms of the fraction of the time spent in failure state by the component. Once the reliability of all components is obtained, a compositional model is used to determine the reliability of specific system configurations. RESIST models the uncertainty in the learning using probabilities.

8.7 POISED

POISED [7] is a quantitative approach for tackling the complexity of automatically making adaptation decisions under uncertainty. It builds on possibility theory and fuzzy mathematics to assess both the positive and negative consequences of uncertainty. The goal in POISED is to improve the quality attributes of a software system through reconfiguration of its components to achieve a global optimal configuration for the software system. POISED redefines the conventional definition of optimal adaptation decision to one that has the best range of behavior. In turn, the selected solution has the highest likelihood of satisfying the system's quality objectives, even if due to uncertainty, properties expected of the system are not borne out in practice. This is different from conventional approaches, which do not incorporate uncertainty in their analysis. Such approaches consider the behavior of the system as a point estimate, while POISED consider a range of behavior.

POISED provides a framework to gather and build up uncertainties into a coherent representation, which lends itself well to decision making. POISED relies on Possibilistic Linear Programming to make the trade-off between different configuration alternatives. The configuration knobs in POISED allow the decision maker to specify what aspect of uncertainty is more important: in some cases a solution capable of providing certain guarantees in the worst case scenario would be desirable, in others a solution with higher risk, but the potential of higher quality may be desirable.

9 Conclusion

Uncertainty is a well-known challenge in the construction of dependable self-adaptive software, yet it is a relatively unexplored topic in this area of research. We believe lack of a coherent understanding of uncertainty has hindered the development of suitable techniques to mitigate it. This in turn has prevented the application of solutions developed and evaluated in the academic settings to real-world software systems that are often risk-averse. We believe for widespread adoption of self-adaptation capabilities in real-world application, the research community needs to first develop suitable and practical mechanisms to control the risk associated with self-adaptation of software under uncertainty.

This paper has aimed to address this issue by shedding light on the role of uncertainty in self-adaptive software and distilling its characteristics. We used a robotic software system to illustrate the impact of uncertainty in process of making adaptation decisions, and proposed an alternative method of reasoning about optimality of adaptation decisions that takes imprecision and variability of the knowledge into account. We also provided an overview of the state-of-the-art approaches that have tackled the different facets of uncertainty in self-adaptive software.

While a series of recent publications in this area of research have provided a good foundation for addressing uncertainty issues in self-adaptation, several research challenges remain. One of the most critical issues is that the majority of mathematical techniques for dealing with uncertainty are computationally very expensive. For instance, the standard operations research technique for making decisions under probability theory is called *stochastic programming*. However, stochastic programming is known to be computationally expensive for execution, which makes it unsuitable for use at runtime, where often decisions have to be made very fast.

Another challenge is the ability to quantify uncertainty, which is necessary to be able to reason about uncertainty and adopt the new definition of optimality advocated in this paper (recall Section 5). This is particularly difficult when the uncertainty is in sources that are not necessarily under the control of self-adaptive software (e.g., uncertainty is in the environment). While generally this is a challenging problem that requires further research, our recent work [7] shows that even if uncertainty can only be partially quantified (i.e., roughly estimated), by incorporating it in the analysis, self-adaptation logic is able to make better choices than if it was to completely ignore uncertainty.

Acknowledgments. This work is partially supported by grant CCF-0820060 and CCF-1217503 from the National Science Foundation and grant N11AP20025 from Defense Advanced Research Projects Agency.

References

1. Kramer, J., Magee, J.: Self-Managed systems: an architectural challenge. In: Int'l Conf. on Software Engineering, Minneapolis, Minnesota, pp. 259–268 (2007)
2. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
3. De Lemos, R., Giese, H., Muller, H.A., Shaw, M., Andersson, J., Baresi, L., Becker, B., Bencomo, N., Brun, Y., Cikic, B., Desmarais, R., Dustdar, S., Engels, G., Geihs, K., Goeschka, K.M., Gorla, A., Grassi, V., Inverardi, P., Karsai, G., Kramer, J., Litoiu, M., Lopes, A., Magee, J., Malek, S., Mankovskii, S., Mirandola, R., Mylopoulos, J., Nierstrasz, O., Pezze, M., Prehofer, C., Schafer, W., Schlichting, W., Schmerl, B., Smith, D.B., Sousa, J.P., Tamura, G., Tahvildari, L., Villegas, N.M., Vogel, T., Weyns, D., Wong, K., Wuttke, J.: Software engineering for Self-Adpaptive systems: A second research roadmap. In: Lemos, R.d., Giese, H., Muller, H., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems, Dagstuhl, Germany (2011)
4. Garlan, D.: Software engineering in an uncertain world. In: FSE/SDP Wrkshp. on the Future of Software Engineering Research, Santa Fe, New Mexico (2010)
5. Aughenbaugh, J.M.: Managing uncertainty in engineering design using imprecise probabilities and principles of information economics. PhD thesis, Georgia Institute of Technology (2006)

6. Cheng, S.W., Garlan, D.: Handling uncertainty in autonomic systems. In: Int'l Wrkshp. on Living with Uncertainty, Atlanta, Georgia (2007)
7. Esfahani, N., Kouroshfar, E., Malek, S.: Taming uncertainty in Self-Adaptive software. In: The Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Szeged, Hungary (2011)
8. Malek, S., Edwards, G., Brun, Y., Tajalli, H., Garcia, J., Krka, I., Medvidovic, N., Mikic-Rakic, M., Sukhatme, G.S.: An architecture-driven software mobility framework. Journal of Systems and Software 83, 972–989 (2010)
9. Menasce, D.A., Dowdy, L.W., Almeida, V.A.: Performance by Design: Computer Capacity Planning By Example. Prentice Hall PTR (2004)
10. Weyns, D., Malek, S., Andersson, J.: FORMS: a formal reference model for self-adaptation. In: Int'l Conf. on Autonomic Computing, Washington, DC, pp. 205–214 (2010)
11. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36, 41–50 (2003)
12. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with reusable infrastructure. IEEE Computer 37, 46–54 (2004)
13. Trouwborst, A.: Precautionary rights and duties of states. Martinus Nijhoff (2006)
14. Cooray, D., Malek, S., Roshandel, R., Kilgore, D.: RESISTing reliability degradation through proactive reconfiguration. In: Int'l Conf. on Automated Software Engineering, Antwerp, Belgium (2010)
15. Winkler, R.L.: Uncertainty in probabilistic risk assessment. Reliability Engineering & System Safety 54, 127–132 (1996)
16. Bertsekas, D.P., Tsitsiklis, J.N.: Introduction to Probability, 2nd edn. Athena Scientific (2008)
17. Kolmogorov, A.: Foundations of Probability (1933)
18. Hoff, P.D.: A First Course in Bayesian Statistical Methods, 2nd printing edn. Springer (2009)
19. Zadeh, L.A.: Fuzzy sets. Information and control 8, 338–353 (1965)
20. Zadeh, L.A.: Fuzzy sets as a basis for a theory of possibility. Fuzzy Sets Syst. 100, 9–34 (1999)
21. Coletti, G., Scozzafava, R.: Conditional probability, fuzzy sets, and possibility: a unifying view. Fuzzy Sets and Systems 144, 227–249 (2004)
22. Dubois, D., Prade, H.: Possibility theory, probability theory and Multiple-Valued logics: A clarification. Annals of Mathematics and Artificial Intelligence 32, 35–66 (2001), ACM ID: 590454
23. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.: RELAX: incorporating uncertainty into the specification of Self-Adaptive systems. In: Int'l Requirements Engineering Conf., Atlanta, Georgia, pp. 79–88 (2009)
24. 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)
25. Baresi, L., Pasquale, L., Spoletini, P.: Fuzzy goals for Requirements-Driven adaptation. In: Int'l Requirements Engineering Conf., Sydney, Australia, pp. 125–134 (2010)
26. Elkhodary, A., Esfahani, N., Malek, S.: FUSION: a framework for engineering Self-Tuning Self-Adaptive software systems. In: Int'l Symp. on the Foundations of Software Engineering, Santa Fe, New, Mexico, pp. 7–16 (2010)

27. Poladian, V., Sousa, J.P., Garlan, D., Shaw, M.: Dynamic configuration of Resource-Aware services. In: Int'l Conf. on Software Engineering, Scotland, UK, pp. 604–613 (2004)
28. Poladian, V., Garlan, D., Shaw, M., Satyanarayanan, M., Schmerl, B., Sousa, J.: Leveraging resource prediction for anticipatory dynamic configuration. In: Int'l Conf. on Self-Adaptive and Self-Organizing Systems, Boston, Massachusetts, pp. 214–223. IEEE Computer Society (2007)
29. Narayanan, D., Satyanarayanan, M.: Predictive resource management for wearable computing. In: Int'l Conf. on Mobile Systems, Applications and Services, San Francisco, California, pp. 113–128 (2003), ACM ID: 1189041

A Software Lifecycle Process to Support Consistent Evolutions

Paola Inverardi[1] and Marco Mori[2]

[1] Dip. di Informatica, Università dell'Aquila
paola.inverardi@di.univaq.it
[2] IMT Institute for Advanced Studies Lucca
marco.mori@imtlucca.it

Abstract. Ubiquitous software systems evolve their behavior at run-time because of uncertain environmental conditions and changing user needs. This paper describes our approach for a model-centric software evolution process of context-aware adaptive systems. Systems are represented following the feature engineering perspective and this modeling supports foreseen and unforeseen evolution. The first one deals with foreseen contexts while unforeseen evolutions address new user needs arising at run-time possibly in response to unforeseen context changes. The main contribution of this paper is a generic software lifecycle process for context-aware adaptive systems that allows systems to be managed both at design time and at execution time by exploiting suitable models. The approach supports both static and dynamic decision-making mechanisms to enact evolutions and to check the evolution consistency.

Keywords: Context-aware adaptive systems, software lifecycle process, variability model, consistent evolution.

1 Introduction

In the era of ubiquitous computing, software systems have to be designed and developed taking into account the information coming from the surrounding environment. This new dimension, called *context*, has to be exploited to make systems flexible and adaptive. Context is not completely known at design time thus making the process of designing and developing ubiquitous applications continue at execution time [13,18,25]. Software engineers define software alternatives having in mind a partial representation of the context in which the system is going to operate. Since it is not always possible to have a complete representation of the environment, the software engineer cannot provide all the software alternatives at design time. In addition resource-constrained devices limit the number of admissible alternatives. Thus the set of software alternatives provided at design time may have to be augmented in order to face new unforeseen environmental conditions.

We consider two systems-related characteristics: *context-awareness* expresses the ability of accessing and exploiting environmental information [7,21,28], and

R. de Lemos et al. (Eds.): Self-Adaptive Systems, LNCS 7475, pp. 239–264, 2013.

adaptivity which makes a system flexible by supporting behavioral variations [47,3,43]. However, systems should evolve in a consistent way with respect to the context in which they operate. Models can play a key role for developing and evolving context-aware adaptive applications since they support the consistent evolution required by context variations. Different models are required to achieve a consistent evolution:

- a model for representing the system and its variability;
- models to represent the context surrounding the system;
- requirement engineering models;
- models representing executable artifacts;
- a software process model for the adaptive application.

All mentioned models should be exploited and managed at run-time when the development of the system is still required [10,44]. Therefore, on one hand models should provide the right level of adaptivity for the system while on the other hand they should be suitable in terms of required computational effort. This means that the time required to accomplish the model-based consistency check should be negligible with respect to the interval between consecutive evolution requests.

Models should be exploited by an integrated support in order to automate, as much as possible, the process of developing and evolving adaptive applications. This would enable the software engineer to reuse a set of "good practice" and tools for building and maintaining such applications [37].

This paper defines a generic software lifecycle process for context-aware adaptive systems. The process we propose supports two kinds of evolution while keeping the system consistent with the context. *Foreseen evolution* addresses foreseen context variations whereas *unforeseen evolution* deals with unforeseen context variations. Figure 1 shows how context affects both system evolutions. In the foreseen evolution the system evolves in order to keep satisfied a fixed set of requirements by switching among different software alternatives provided at design

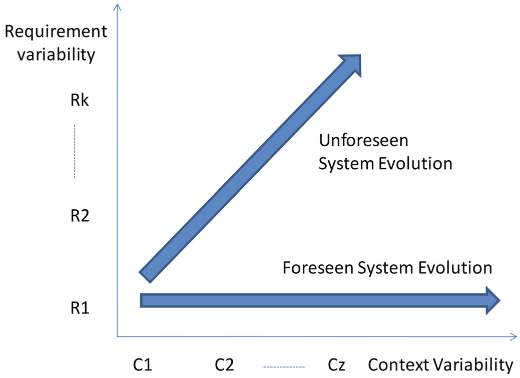


Fig. 1. System evolutions

time for different known contexts. In the unforeseen evolution the system evolves to satisfy changing user needs (requirements) that may emerge as a consequence of an unforeseen context variation. The system evolves by switching to a new un-anticipated software alternative whose behavior includes new functionality necessary to satisfy the emerging requirement. We represent the system following the Software Product Line Engineering (SPLE) perspective since it breaks the system complexity into feature components thus reducing the impact that any change may have on the system [29]. In addition the SPLE perspective already provides models to manage the system and to support consistent evolution.

This paper extends preliminary work [22] on the use of feature engineering for modeling the evolutions of context-aware adaptive systems. Our process is amenable to develop and evolve highly-configurable systems with interfering features. It exploits the SPLE perspective in order to provide a uniform abstraction to all the development approaches that consider a system to be made out of a combination of basic software entities, such as the Component Off The Shelf (COTS) approach or the service-oriented one. We assume that we have a set of basic behavioral elements as input to our software process. These basic elements will contain implementation artifacts with corresponding requirements specifications. It is worth stressing that while we do address the problem of managing model-centric evolution we will only briefly comment on the mechanisms to enact the evolution. We will assume that the system may be reconfigured when it is in a state in which the evolution is allowed (e.g. quiescent state or weaker notions [30,46]).

In this paper we also define a generic evolution framework to support our software process in performing its run-time activities. To this end, the framework implements a control loop to monitor and to evolve the adaptive application.

The contributions of this paper are:

- a process methodology to design and develop context-aware adaptive applications;
- a set of models to represent the system and the context along with their evolutions;
- a methodology to check the consistency based on the context;
- an architecture implementing the support for the evolutions.

We will explain our approach by means of an adaptive application which elaborates a Mandelbrot fractal [33] that better fits the characteristics of the mobile device (CPU, memory, number of display colors,...). The application requirements consist in visualizing a fractal image to the user through the device screen. The higher the level of context resources available, the more beautiful will be the fractal image shown to the user. The fractal context-aware adaptive system will be modeled through a set of features which represent the basic alternative behaviors to color, build and view the fractal image. At run time features may need to be activated or de-activated based on the context-resource availability changes. In addition because of environment unpredictability, the user may want to introduce new unforeseen behavior as the system operates in an unforeseen context. For example whenever the unforeseen device characteristics makes the visualization of the fractal image format impossible, the user may guide the introduction of a new software plug-in to decode that specific format.

The remainder of this paper is structured as follows. Section 2 describes related work to address system evolutions while Section 3 introduces the basic models of our generic evolution framework. In Section 4 we define the software lifecycle process to design and develop context-aware adaptive applications. Section 5 describes how our process supports foreseen and unforeseen evolutions. Section 6 proposes the interface architecture which implements the generic evolution framework along with a possible instance with current practice technologies. Section 7 provides a summary of our contribution and a discussion of future work.

2 Related Work

In the literature several frameworks address system evolution. They exploit models with different granularity such as context-aware requirements models, context-aware architectural models and context-aware implementation models.

The Rainbow framework [17] enables architectural self-adaptation by exploiting predefined adaptation rules. System components are reconfigured based on decisions taken at design time while no un-anticipated adaptations can be achieved. The framework supports non-functional reconfigurations while it does not consider the consistency checking of the evolution. The context is not explicitly modeled but simple variables are considered in the framework.

The PLASTIC approach [4] applies reconfigurations at the implementation level by exploiting an explicit definition of context model. The approach supports non-functional reconfigurations of statically defined Java artifacts driven by context variations. The framework only deals with foreseen evolution while run-time evolution is not allowed. The Javeleon framework [20] as well as the JavAdaptor framework [41] aims to support the run-time evolution by means of transparent dynamic updates of running Java applications. Developers can simply evolve their applications at run-time and they can trigger an on-line update without stopping the running application. Javeleon and JavAdaptor do not support a definition of context for the evolution but the developers is directly in charge of injecting new behaviors in the application at run-time. These approaches as well as the PLASTIC framework do not provide a process to assess the consistency of foreseen and unforeseen evolution. Ali et al. [2] propose a goal-based framework to enact the evolution among system variants at requirement level. This approach provides a context analysis phase to discard variants that are inconsistent based on the context predicates. Nevertheless, it only supports the design-time analysis on the contextual goal model. Qureshi and Perini [42] have defined a framework for requirement engineering to distinguish activities at design-time from activities at run-time. They have provided a mechanism to evolve the requirement specification at run-time driven by the user thus supporting a notion of unforeseen evolution. Nevertheless the proposed method is not applied to a real case study and no definition of consistency is considered in the framework. Kramer and Magee [31] have presented a three-layered conceptual model to support the architectural reconfiguration of self-managing systems. They consider a Component layer, a Change management layer and a Goal management layer. The Goal layer identifies the plan to execute while the Change layer enacts the plan execution by interacting with the Component layer. This feature supports reconfigurations required by new requirements arising at run-time, i.e. unforeseen evolutions. The framework provides functional and non-functional evolutions but it lacks a definition of consistency checking for the composition of components.

To the best of our knowledge the frameworks presented in the literature only apply reconfigurations at specific granularity levels, either at requirements models, or at architectural models or at implementation models. Only a few of them support evolution at run-time while there is almost no support to check the consistency of the evolution. In order to provide high-assurance for context-aware adaptive applications it is necessary to support a definition of consistency as proposed by Zowghi and Gervasi [48]. They suggest that an effective support to consistency is based on system models at the different granularity levels, ranging from the problem space models to the solution space models. We claim that adaptive applications are not developed and evolved following a software process which considers all these models together thus making it difficult to effectively support the consistency of the evolution.

3 Evolution Framework

The evolution framework we propose is characterized by different building blocks to represent the system along with its variability. The system is represented by *units of behavior* which are composed through a *feature diagram* into different *system configurations*. The *context model* enacts the switching process among configurations and it supports the *consistency checking* process for the evolution.

Our evolution framework implements a MAPE (Monitoring Analyze Plan and Execute) cycle in order to support the supervision, execution and evolution of adaptive applications [12]. In Figure 2 we show how the framework implements each of the four phases.

A *monitoring* phase activity collects information from the environment and from the user in order to establish if evolution is required or not. On the one hand, foreseen context variations and user preferences variations may both enact foreseen evolutions. The first influences the admissibility of the system variants, whereas the second influences the fitness of the system variants. On the other hand, unforeseen context variations may force the user to introduce a new requirement into the running variant.

The *analyze* phase determines if the variant to adopt is consistent or not. In case of foreseen evolution we consider the set of system variants that are consistent at the current context state. The consistency at each different context state is proven at design time. In case of unforeseen evolution the analysis is

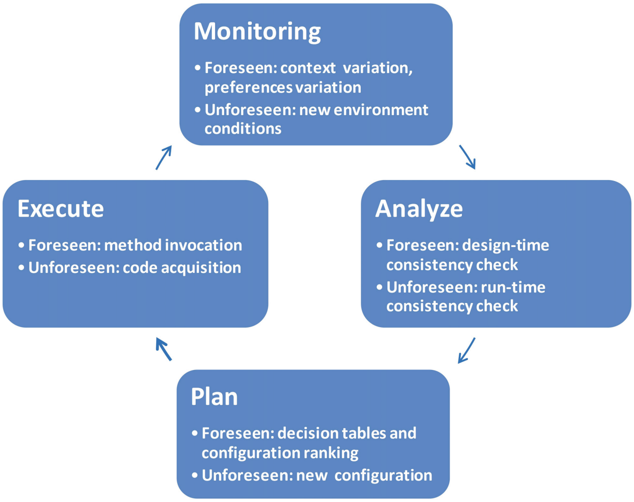

Monitoring
• Foreseen: context variation, preferences variation
• Unforeseen: new environment conditions
Analyze
• Foreseen: design-time consistency check
• Unforeseen: run-time consistency check
Plan
• Foreseen: decision tables and configuration ranking
• Unforeseen: new configuration
Execute
• Foreseen: method invocation
• Unforeseen: code acquisition

Fig. 2. MAPE cycle

performed at run-time by checking the consistency for the un-anticipated variant. This variant will contain the same set of features as the current variant plus a new feature that implements the new requirement specified by the user.

After the analyze phase the *planning* phase supports the decision-making process for the variant to adopt. For the foreseen evolution, a ranking mechanism establishes the most suitable variant based on context and user preferences. For the unforeseen evolution the new variant which has been proven consistent at the analyze phase is put forward to the execution phase.

Finally at the *execution* phase the system switches from the current to the target variant. The target variant is enacted through its entry point method. For the unforeseen evolution it is also necessary to incorporate a new code artifact into the target variant before enacting its execution.

In the following we define the elements of our approach before describing the software development process for context-aware adaptive systems in Section 4.

3.1 Context Model

In our approach the context model expresses the set of external entities that are beyond the system's control but which may influence the system execution. Our context model consists of key-value pairs and it is defined using two perspectives: the *context structure* and the *context space*. The *context structure* expresses resources in term of types and categories. We adopt the resource taxonomy where each context element belongs either to the system, to the user or to the physical environment. In addition we consider the resource types enumerate, boolean and natural [32]. In Figure 3 is depicted a context structure which conforms to the meta-model shown at the left side of the figure.

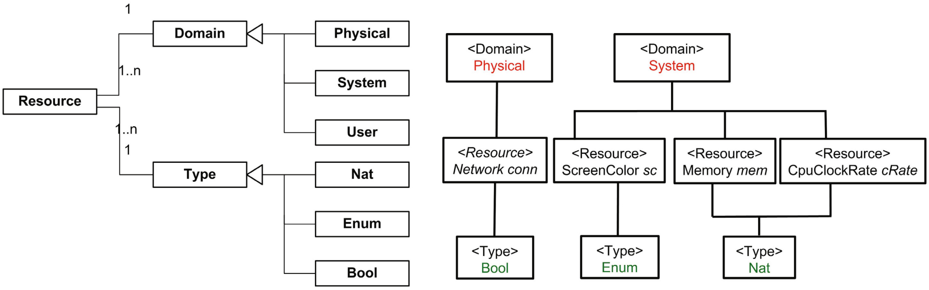


Fig. 3. Context structure: meta-model and model

The *context space* expresses the variability for the resource assignment. Each resource is identified through a tag $ResId$ and it can assume one among its admissible values contained in $dom(ResId)$. The context space for the resources $ResId_1, ..., ResId_n$ is defined as the Cartesian product:

$$S = \bigotimes dom(ResId_i) \quad s.t. \quad i = 1, ..., n \tag{1}$$

Each valid assignment of resources $\overrightarrow{c} \in S$ will be considered as a different context state. Let us consider four different resources respectively expressing the free memory, the CPU clock rate, the number of screen colors and the network availability (0 if false, 1 if true): $dom(mem) = \{100, 150, 250, 350\}$, $dom(cRate) = \{200, 400, 600\}$, $dom(sc) = \{256, 4096\}$, $dom(conn) = \{0, 1\}$. The context model space will be composed of $4 * 3 * 2 * 2 = 48$ states.

3.2 Unit of Behavior

In our vision we represent context-aware adaptive applications in terms of sets of dynamic units called features. A feature is the smaller part of a service that can be perceived by the user. We define a feature by a context-independent requirement, a context-dependent constraint requirement, and an implementation part. The notion of requirements we adopt follows the taxonomy proposed by Glinz [19]. The definition of requirements is based on the *concern* to which a requirement pertains. Given that a concern is a matter of interest in a system, the taxonomy considers functional requirements which pertain to functional concerns, performance requirements which pertain to performance concerns and specific quality requirements which pertain to quality concerns. In addition constraint requirements limit the solution space of functional, performance and specific quality requirements. We adopt this taxonomy and we exploit the feature definition in [14] in order to propose our definition of a feature as a triple $f_i = (R_i, I_i, C_i)$, where:

- R_i is a conjunction of functional, performance and specific quality requirements (context-independent); an example of a functional requirement is:

Compute and visualize each fractal pixel. A quality requirement is: *The image is visualized a pixel at a time* which in terms of implementation consists in assigning the value *Immediate* to the quality property *DisplayModel*.

- I_i is the the component/service implementing the feature. It is expressed as Java code, e.g. see Figure 4.

```
public class MandelCanvas{
 ...
 public void generateImmediateFractal(){
  image = Image.createImage(width, height);
  Graphics imageGraphics_DirectCanvas = image.getGraphics();
  for (int x = 0; x < width; x++) {
   for (int y = 0; y < height; y++){
    FractalPixel pixel_DirectCanvas = drawFractalPixel(x,y);
    repaint();
   }}
}...}
```

Fig. 4. Example: feature implementation

- C_i is a context-dependent constraint requirement defined as a predicate over the context entities, e.g. *mem* $\geq$ *120kb*.

3.3 System Configuration

A *system configuration* is obtained assembling a set of features. Each configuration expresses the set of functionalities that a system shows to a user at a certain step of the evolution. Given the set of features F, a system configuration is a triple obtained combining each feature in F as $G_F = (R_F, I_F, C_F)$. At this level of description we do not explain how to combine features. We just suppose to have an abstract union operator among features which is defined in terms of union operators for context-independent requirements, context-dependent requirement and implementation components. The actual implementation of these union operators will depend on the specific formalisms that may be used to express these three elements. Given two features $f_1 =< R_1, I_1, C_1, >$ and $f_2 =< R_2, I_2, C_2 >$ their union is defined as: $f_1 \cup_f f_2 =< R_1 \cup_R R_2, I_1 \cup_I I_2, C_1 \cup_C C_2 >$. In the following we show a possible example on how to merge context requirements and implementation components for a system configuration starting from its features.

The union operator $\cup_C$ merges context requirements depending on the nature of resources. For example if we have two requirements demanding bandwidth for 20 kbps each one, their union will express a demand of bandwidth for 40 kbps. For the implementation portion I the software engineer combines the code artifacts in order to have a single access point to the whole configuration. Each configuration is composed by a Java class for each single feature plus a Java class which is the entry point for the system configuration. This class entails the method *execute* to trigger the execution of the configuration (e.g. see Figure 7).

Given a system definition it is necessary to model in which different future system configurations the system may evolve. The variability model that we have chosen is inspired by the *feature model* which has been first introduced in the Feature-Oriented Domain Analysis method [27]. Since then, feature modeling has been widely adopted by the SPL engineering community and a number of extensions have been proposed [45]. In our approach we consider a possible abstract syntax for the feature model defined starting from nodes (features) and arcs between nodes:

- The root node of the model is the label which stands for the system.
- Each node expresses a feature which can be either optional or mandatory.
- Each edge between two nodes expresses a decomposition relation (consist-of) between the parent node and the child node. It enables the possibility to add behavior to the parent feature. We consider two decomposition relations: AND decomposition and XOR decomposition.
- "Requires" constraint is a directed relation between two features. If one feature is present in the configuration the second has to be present as well.
- "Mutex" constraint enables the mutual exclusion between two features; therefore they cannot be in the system configuration simultaneously.

Starting from the feature model (abstract syntax), the feature diagram (concrete syntax) is commonly expressed as a tree structure. We adopt a subset of the syntax presented in [15]. In this diagram, features are represented in a tree-like format. Dark circles represent mandatory features, while white circles represent optional features. An inverted arc among multiple arcs expresses a XOR decomposition meaning that exactly one feature can be selected. Multiple arcs that start from a parent node express an AND decomposition.

Starting from the feature diagram, the set of possible system configurations is obtained by combining the features in subsets compliant to the diagram. The diagram shown in Figure 5 concerns our case study and contains 8 features which give rise to 10 system configurations. Each configuration contains only one feature to generate the image and only one feature to color it. An admissible configuration contains the features to download a predefined image from a remote server. We further discuss the features of the fractal application in Section 4.1.

Each system configuration is mapped to its implementation which enables its execution. Let us consider the system configuration $G_4 = \{f_{genPro}, f_{colB}\}$ implemented by the class diagram depicted in Figure 6. Each feature in G_4 is implemented as a single Java class. The class $MandelCanvas$, which implements f_{genPro}, provides the interface $generateProgressiveFractal$ that generates and draws the fractal image progressively a row at a time exploiting the operation $drawFractalPixel$. The class $Colouring$, which implements f_{colB}, provides the interface $pixelColourAsBands$ and the operation $initColourAsBands$ in order to color the image with different bands of colors. The only class which does not correspond to any of the features within the configuration is $LocalFractalApp$ that is the external interface to access the whole application variant. The configuration is enacted through its method $execute$ which implements the logic of the variant. Figure 7 shows an excerpt of the Java specification for configuration G_4.

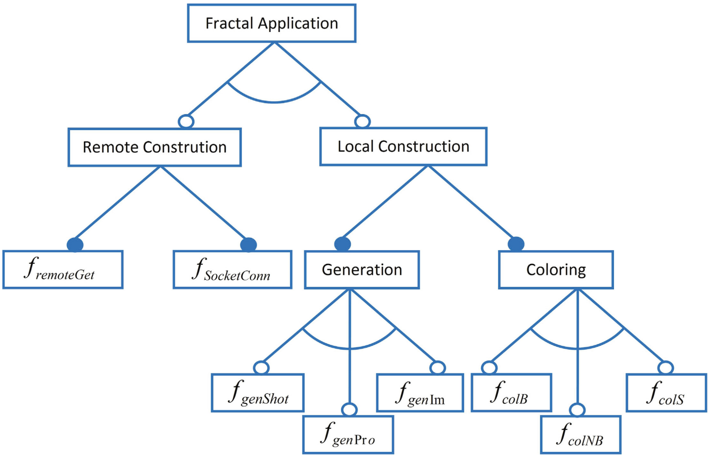


Fig. 5. Feature diagram

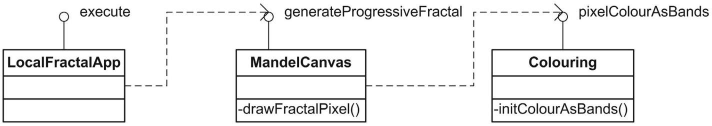


Fig. 6. Example: class diagram (G_4)

3.4 Consistency Checking

We propose a notion of consistency that is based on the notion of *feature interaction*. A feature interaction occurs when two or more features run correctly in isolation but they give rise to undesired behavior when jointly executed [1,9,38]. A certain system configuration is *consistent* if its features does not give rise to any feature interaction phenomenon. Following the Problem Frame approach [26] as exploited in [14], we formalize our notion of consistency for a certain configuration $G = (R_F, I_F, C_F)$ as:

$$I_F, C_F \vdash R_F \tag{2}$$

This definition entails three different problems:

- $(C_F)[\overrightarrow{c}/\overrightarrow{x}]$: this formula checks the joint context requirement (predicate) C_F assigning the current context values $\overrightarrow{c}$ to the formal parameters $\overrightarrow{x}$;
- R_F *is Satisfiable*: this formula checks if the joint context-independent requirement can be satisfied;
- $I_F \vdash R_F$: this formula validates the joint implementation with respect to the joint requirement either by means of model checking or through a testing process.

```
public class LocalFractalApp extends MIDlet {
 MandelCanvas mandelCanvas;
 ...
 public LocalFractalApp(){
  mandelCanvas = new MandelCanvas();
 }
 protected void execute(){
  currentDisplay = Display.getDisplay(this);
  currentDisplay.setCurrent(mandelCanvas);
  mandelCanvas.generateProgressiveFractal();
  exitAction();
 } ... }
public class MandelCanvas extends Canvas {
 ...
 public void generateProgressiveFractal(){
  int column_ArrayCanvas[] = new int[height];
  for (int x = 0; x < width; x++){
   for (int y = 0; y < height; y++){
    FractalPixel pixel_ArrayCanvas = drawFractalPixel(x, y);
    column_ArrayCanvas[y] =  pixelColor(pixel_ArrayCanvas.isInsideFractal(),
    pixel_ArrayCanvas.getIterations(),pixel_ArrayCanvas.getDistance());
   }
   offsetX = x;
   image = Image.createRGBImage(column_ArrayCanvas, 1, height, false);
   repaint();
 }} ... }
public class Colouring{
 ...
 private int pixelColourAsBands(boolean interno, int iterazioni,
 double dist){
  int tmp= (interno ? 0 : colors[iterazioni % paletteNumColors]);
  return tmp;
 }
 private void initColourAsBands(){
  int[] tmpColors_IterationsLimitedPalette = { −256, −16711681,−65281,−256,
  −4194304, −16728064, −16777024, −8323073, −32513, −128 };
  colors = tmpColors_IterationsLimitedPalette;
  paletteNumColors = colors.length;
 }... }
```

Fig. 7. Example: implementation (G_4)

Since our aim is to support a notion of consistency that can be performed at run-time we should take into account the computational effort for the corresponding three algorithms. Among them, checking the context requirement against a certain context state is the less expensive in terms of time and space. Although it is only a necessary but not sufficient condition for a complete notion of consistency it plays a key role to check the consistency of ubiquitous applications. Indeed, the serendipity of the environment that characterizes this kind of systems makes them very vulnerable to context variations. Therefore a weak notion of consistency can be based only on context requirements satisfiability:

Definition 1. *G* is *weakly consistent* in $\overrightarrow{c}$ iff $C_G[\overrightarrow{c}/\overrightarrow{x}]$ is True

Let us consider the configuration $G_x = \{f_{getRem}, f_{sockConn}, f_{tiffViewer}\}$ where each feature is characterized by the correspondent context requirement:

(i) $C_{tiffViewer} ::= \mathit{cRate} \geq 300 \wedge \mathit{mem} \geq 35$
(ii) $C_{getRem} ::= \mathit{mem} \geq 100$
(iii) $C_{sockConn} ::= \mathit{conn} = 1$

We assume that G_x has to be checked at the context state $\overrightarrow{c} = (100, 300, 4096, 1)$. This state provides 100 Kb of memory, a CPU clock rate of 300 Mhz, a screen device with 4096 colors and an Internet connection. Although each context requirement is weakly consistent separately at $\overrightarrow{c}$, the whole configuration is not weakly consistent because of the limited availability of free memory. Indeed, if we combine the request of memory coming from the context requirement (i) and (ii) we obtain a total request for $135Kb$ of memory that cannot be satisfied at the context state $\overrightarrow{c}$. Therefore it is not possible to execute the features $f_{getRem}, f_{sockConn}$ and $f_{tiffViewer}$ together at $\overrightarrow{c}$.

In our previous paper [23] we have extended this notion of weak consistency by defining a mechanism to check the configuration requirements with respect to the implementation artifacts. This enables us to catch also interactions that arise at the code level.

4 Software Development Process

In this section we describe how we support the development of a context-aware adaptive application. We have defined a software lifecycle process which follows the structure presented by Autili et al. [6]. Our software process implements four different activities, namely Explore, Integrate, Validate and Evolve as shown in Figure 8. The *exploration* phase exploits a feature library containing the code implementation and the correspondent requirements descriptions. The *integration* phase takes these features as input and it produces the space of the system variants as a feature diagram. Each variant is checked though a validation phase which performs the context analysis [24] and model checking [23]. Finally, the *evolution* phase reconfigures the system by switching from the current configuration to the new one.

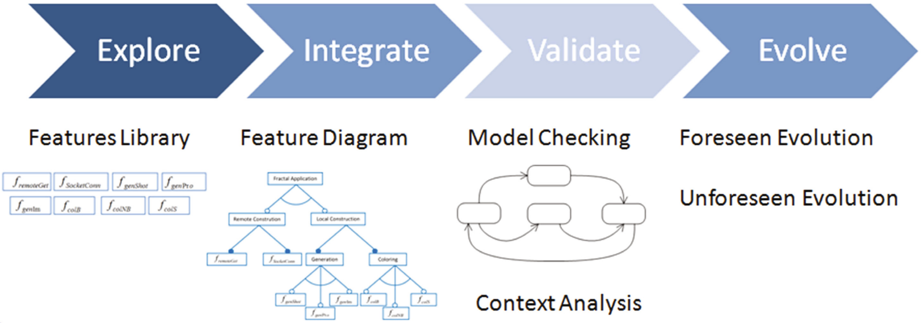


Fig. 8. Software process

In the remainder of this section we describe how our generic evolution framework supports the software process. The problem we face is the complexity for the software engineer to specify the context conditions under which each system configuration is admissible. Given n features it could be required to set the context conditions for 2^n configurations in the worst case. Our methodology makes the generation of the system configuration automatic by exploiting the models provided in the SPLE as described in Section 3.

At the *exploration* phase the software engineer defines the set of features of interest. Starting from a standard component it is possible to define a feature $f = (R, I, C)$ by considering the requirements of the component and its code. The feature code will be exactly the same as the code of the component. R will contain the requirements of the component that are not context-dependent. In general the requirements of the component will always contain requirements about the execution context, thus they will be added to C. Further context requirements, for example concerning resources consumptions can be obtained through suitable static code analysis. For example in our environment we use the Chameleon framework [5] in order to extract the consumption of resources caused by I (e.g. memory and CPU clock rate). At the end of the exploration phase we obtain a set of features defined in terms of their basic components, i.e. $A = \{f_1, .., f_n\}$.

At *integration* phase the software engineer combines the features in A through the feature diagram definition. Architectural constraints will be defined here at the integration phase. Starting from the feature diagram an automatic process generates all the system configurations:

$$G = \{G_1, G_2, ..., G_m\} \text{ s.t. } m \leq 2^{|A|} \tag{3}$$

We assume that the requirements belonging to each configuration imply the system requirements. We further assume that each configuration satisfies its requirements: $I_{G_i} \vdash R_{G_i}\ \forall i = 1, .., m$. An automatic process generates the context structure and the context space S considering the context entities exploited by the context requirements belonging to the created configurations.

At *validation* phase we create the data structure to support the evolution. This phase takes place by means of two main steps. The first step consists in labeling each context state $\overrightarrow{c}$ in S with all the features which are consistent in $\overrightarrow{c}$ (Eq. 2). The *feature consistency table* is built inserting value 1 each time a feature is consistent in the corresponding context state. The second step consists in labeling each context state $\overrightarrow{c}$ in S with all the system configurations that are consistent in $\overrightarrow{c}$. The *configuration consistency table* is built inserting value 1 each time a configuration is consistent in the correspondent context state. Finally, we aggregate the context states that make the same set of configurations consistent. Nevertheless, we do not address the scalability problems arising from the number of context states and configurations within the mentioned tables. Different approaches [11] have been presented to reason about the configurations belonging to the feature diagram. Moreover the exponential growth of context states could be mitigated by clustering the states [16].

The *evolution* phase reconfigures the system whenever either a foreseen or an unforeseen evolution is required. In the first case we query the configuration consistency table to retrieve the space of the admissible configurations. Among them we select the most suitable one based on the data structures provided at the validation phase. In the second case we have to re-iterate the first three phases of our software process in order to evolve the system. We query a remote feature library to retrieve the feature implementing the new requirement and we integrate the new feature with the current configuration. Finally, we have to validate the new unforeseen configuration before we can add it to the configuration consistency table. The evolution processes are further discussed in Section 5.

4.1 Working Example

In this section we show how we design and develop the adaptive application to visualize a Mandelbrot fractal. To this end, the software engineer defines the set of features A in terms of requirements and code implementations:

$$A = \{f_{genShot}, f_{genPro}, f_{genImm}, f_{colB}, f_{colNB}, f_{colS}, f_{remGet}, f_{sockConn}\}$$

The set A contains the features to generate and color the fractal pixels and the features to download a standard fractal image from a remote server. The generation may be performed by visualizing a pixel at a time (f_{genImm}), a pixel row at a time (f_{genPro}) or the whole fractal image at the end of the drawing process ($f_{genShot}$). The pixel colors are defined following three different schemas: f_{colB} colors pixels as bands exploiting a limited number of tones; f_{colNB} colors pixels as bands exploiting a wide spectrum of tones while f_{colS} follows a smooth schema to color pixels exploiting a wide spectrum of tones. Finally, $f_{sockConn}$ connects the device to the Internet whereas f_{remGet} retrieves and views a standard fractal image from a remote server.

Figure 9 shows an excerpt of the features entailed in A. It is possible to define the context requirement of each feature by exploiting the Chameleon framework in order to obtain the consumption of resources, e.g. CPU clock rate and memory. Further, context requirements can be defined by extracting the requirement on the number of screen colors derived from the requirement of the component.

In order to design the fractal application the software engineer combines the features and produces the feature diagram as shown in Figure 5. The logic operators in the feature diagram guide the automatic generation of 10 system configurations as shown in Table 1. The first nine configurations are obtained combining the three different building mechanisms with three different coloring schemas. The last one simply gets an already defined fractal image from a remote server. Each configuration is characterized by the context requirement and by the offered qualities. The *DisplayModel* quality represents the modality of showing the fractal while *ColorModel* quality expresses the coloring modalities.

After creating the configurations, the integration phase generates the context model which contains the relevant resources for the fractal application as shown in Figure 3. In our example the context space will be defined as $S = mem \times cRate \times sc \times conn$.

$\mathbf{f_{genPro}} = (\mathbf{R_{genPro}}, \mathbf{I_{genPro}}, \mathbf{C_{genPro}})$

$\mathbf{R_{genPro}}$: Compute each fractal pixel and show it a pixel row at a time

$\mathbf{I_{genPro}}$:

```
public class MandelCanvas extends Canvas{ ...
 public void generateProgressiveFractal(){
  int column_ArrayCanvas[] = new int[height];
  for (int x = 0; x < width; x++){
   for (int y = 0; y < height; y++){
    FractalPixel pixel_ArrayCanvas = drawFractalPixel(x, y);
   }
   offsetX = x;
   image = Image.createRGBImage(column_ArrayCanvas,1, height, false);
   repaint();
  } } ... }
```

$\mathbf{C_{genPro}} : \mathbf{mem} \geq \mathbf{200}$

$\mathbf{f_{colS}} = (\mathbf{R_{colS}}, \mathbf{I_{colS}}, \mathbf{C_{colS}})$

$\mathbf{R_{colS}}$: Paint the fractal pixels as smoothly nice colored bands

$\mathbf{I_{colS}}$:

```
public class Colouring{...
 private int pixelColorSmoothly(boolean interno, int iterazioni, double dist){
  if (interno) return 0;
  iterazioni = iterazioni + 2;
  double mu_IterationsDistance = iterazioni−
  (Float11.log(Float11.log(dist))) / log2;
  int tmp= DBL_ToRGB(mu_IterationsDistance);
  return tmp;}
 private void initColorsSmoothly() {
  log2 = Float11.log(2.0);
 }...}
```

$\mathbf{C_{colS}} : \mathbf{crate} \geq \mathbf{500} \wedge \mathbf{sc} \geq \mathbf{4096}$

$\mathbf{f_{remGet}} = (\mathbf{R_{remGet}}, \mathbf{I_{remGet}}, \mathbf{C_{remGet}})$

$\mathbf{R_{remGet}}$:Retrieve and view the fractal image from the server

$\mathbf{I_{remGet}}$:

```
public class RemoteViewer extends Canvas{...
  public void viewRemoteFractal(){
   this.image = getFractal(startTime*1000,maxExecutionTime);
   repaint();
  } ...}
```

$\mathbf{C_{remGet}} : \mathbf{mem} \geq \mathbf{100}$

Fig. 9. Application features

As far as the validation of the fractal application is concerned we only show the consistency based on context analysis (Def. 1). The validation phase creates the feature consistency table (Table 2) by checking the weak consistency for each feature at each context state in S. It evaluates the validity for the context requirements (predicates) of each feature by assigning all the possible context values. The table assigns value 1 if it is possible to select a feature in a certain context state and 0 otherwise. After defining the feature consistency table the context analysis phase creates the configuration consistency table (Table 3) by considering the features included in each configuration. This table contains value 1 only if all the features in a certain configuration are jointly weakly consistent at a certain context state. The process checks the validity of the joint predicate as shown in Section 3.4.

Table 1. System configurations

System Configuration	Context Requirement	Offered Quality
$G_1 = \{f_{genShot}, f_{colB}\}$	$mem \geq 300 \wedge cRate \geq 100$	$DisplayModel = Shot$ $ColorModel = BandOfColors$
$G_2 = \{f_{genShot}, f_{colNB}\}$	$mem \geq 300 \wedge cRate \geq 300 \wedge$ $sc \geq 4096$	$DisplayModel = Shot$ $ColorModel = NiceBandOfColors$
$G_3 = \{f_{genShot}, f_{colS}\}$	$mem \geq 300 \wedge cRate \geq 500 \wedge$ $sc \geq 4096$	$DisplayModel = Shot$ $ColorModel = SmoothyBandOfColors$
$G_4 = \{f_{genPro}, f_{colB}\}$	$mem \geq 200 \wedge cRate \geq 100$	$DisplayModel = Progressive$ $ColorModel = BandOfColors$
$G_5 = \{f_{genPro}, f_{colNB}\}$	$mem \geq 200 \wedge cRate \geq 300 \wedge$ $sc \geq 4096$	$DisplayModel = Progressive$ $ColorModel = NiceBandOfColors$
$G_6 = \{f_{genPro}, f_{colS}\}$	$mem \geq 200 \wedge cRate \geq 500 \wedge$ $sc \geq 4096$	$DisplayModel = Progressive$ $ColorModel = SmoothyBandOfColors$
$G_7 = \{f_{genImm}, f_{colB}\}$	$mem \geq 120 \wedge cRate \geq 100$	$DisplayModel = Immediate$ $ColorModel = BandOfColors$
$G_8 = \{f_{genImm}, f_{colNB}\}$	$mem \geq 120 \wedge cRate \geq 300 \wedge$ $sc \geq 4096$	$DisplayModel = Immediate$ $ColorModel = NiceBandOfColors$
$G_9 = \{f_{genImm}, f_{colS}\}$	$mem \geq 120 \wedge cRate \geq 500 \wedge$ $sc \geq 4096$	$DisplayModel = Immediate$ $ColorModel = SmoothyBandOfColors$
$G_{10} = \{f_{remGet}, f_{sockConn}\}$	$mem \geq 100 \wedge conn = 1$	$DisplayModel = Shot$ $ColorModel = BandOfColors$

Table 2. Feature consistency table

$C(mem, cRate, sc, conn)/f_j$	$f_{genShot}$	f_{genPro}	f_{genImm}	f_{colB}	f_{colNB}	f_{colS}	f_{remGet}	$f_{sockConn}$
$C_0 = (100, 200, 256, 0)$	0	0	0	1	0	0	1	0
...	...	...	...	...	...	...	...	...
$C_{33} = (150, 400, 4096, 1)$	0	0	1	1	1	0	1	1
...	...	...	...	...	...	...	...	...
$C_{43} = (350, 200, 4096, 1)$	1	1	1	1	0	0	1	1
...	...	...	...	...	...	...	...	...
$C_{47} = (350, 600, 4096, 1)$	1	1	1	1	1	1	1	1

Table 3. Configuration consistency table

$C(mem, cRate, sc, conn)/G_k$	G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8	G_9	G_{10}
$C_0 = (100, 200, 256, 0)$	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...
$C_{33} = (150, 400, 4096, 1)$	0	0	0	0	0	0	1	1	0	1
...	...	...	...	...	...	...	...	...	...	...
$C_{43} = (350, 200, 4096, 1)$	1	0	0	1	0	0	1	0	0	1
...	...	...	...	...	...	...	...	...	...	...
$C_{47} = (350, 600, 4096, 1)$	1	1	1	1	1	1	1	1	1	1

5 System Evolution

Our development process supports the system evolution required by the context variations. In the following we show that in the foreseen evolution system and context models are queried to support the reconfigurations, while in the case of unforeseen evolution the same models may have to be refined as a consequence of incoming user needs.

5.1 Foreseen Evolution

In the foreseen evolution we consider only configurations that have already been proven weakly consistent. A monitoring process notifies the context variations which invalidate the context requirement belonging to the running configuration. Whenever such a new assignment of resources is discovered, the framework queries the configuration consistency table to get the possibly new admissible configurations. In order to perform the static decision-making process among weakly consistent configurations we take into consideration context and user preferences. Since we want to make our mechanism resilient to future contexts we take into consideration which is the probable future evolution for each context state. We consider the predictions for the user centric information (user task, user mobility) and the predictions for the evolution laws of resources obtained as explained in [40]. Exploiting such information we build a probabilistic automaton according to the approaches in [34,8]. Each different state corresponds to a different context and each arc expresses the probability to move from a context to another (e.g. Figure 10).

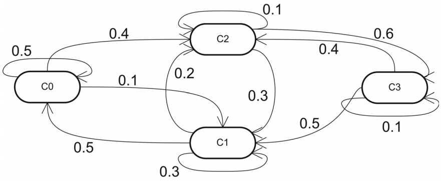


Fig. 10. Probabilistic evolution automata

If the user preferences are not fixed but they change over the execution we can include the possible preference variations within the automaton. Then we exploit the probabilistic model to evaluate the degree of suitability of each configuration to the context and to the user preferences. In [36] we have formalized and we have experimented a decision mechanism process that considers both factors within our probabilistic model.

5.2 Working Example

In the following we show a possible decision-making process that considers fixed user preferences and probable context evolutions in order to evaluate the overall fitness of each configuration G_i. Starting from the automaton in Figure 10 we evaluate the steady-state probability vector $\overrightarrow{p} = [0.2794\ 0.2794\ 0.2647\ 0.1765]$

which expresses how often the context belongs to a certain state. Then we obtain the *context fitness vector* by multiplying the vector $\overrightarrow{p}$ with the matrix m representing the configuration consistency table:

$$f = p \cdot m \tag{4}$$

This vector assigns a fitness value at each configuration that depends on the number of states in which the configuration is admissible and on the relevance for the states as evaluated by the steady-state probability vector. This ranking mechanism considers only how often the context belongs to a certain state whereas it ignores which is the current state and its future transitions thus leading to globally optimum solutions. Parallel to f we also evaluate a *user fitness vector t* expressing how each configuration is suitable with respect to the user preferences. We express preferences as weights over the quality attributes which characterize the variants. Each weight w_q (from 0 to 1) indicates the interest for the user towards a certain quality q. We use a predefined utility function $u_q(G_i)$ to assign a value from 0 to 1 at each quality dimension q provided by each G_i. The software engineer defines the utility functions and the weights for each quality since they are strictly application dependent. The *user fitness vector* is evaluated as:

$$t(G_i) = \sum_{q \in Qualities} w_q \times u_q(G_i) \tag{5}$$

Our decision-making process will consider together the user fitness $t(G_i)$ and context fitness $f(G_i)$ to evaluate the overall fitness of each configuration G_i.

Let us consider the scenario as depicted in Table 3 and let us suppose that the configuration $G_4 = \{f_{genPro}, f_{colB}\}$ is running at the context state $C_{43} = (350, 200, 4096, 1)$ whereas the user preferences assign higher weight to the $DisplayModel$ quality. The system is producing a fractal image drawing a row at a time and coloring pixels as bands of colors. Let us now suppose that because of a new application started on the mobile device, the current memory availability changes and the monitoring detects a context variation. By looking at the new context state $C_{33} = (150, 400, 4096, 1)$ in Table 3 we obtain the set of admissible (weakly consistent) configurations. Among them we select the one with the highest overall fitness. Therefore the current fractal application is stopped and it is evolved towards the configuration $G_7 = \{f_{genImm}, f_{colB}\}$ which represents the best trade-off between user and context fitness.

5.3 Unforeseen Evolution

Let us assume that during the execution phase the set of requirements the system needs to satisfy evolves because of changing user needs. For example the user has to deal with a new context situation that has not been foreseen by the software engineer at design time. Since a new behavior may have to be injected into the system it is necessary to modify at run-time the context-based decision table presented in the earlier sections. In addition also the models related to the system variability and context may have to be refined at run-time. Two different cases can arise: either a new requirement has to be added to the current configuration or an already existing requirement has to be deleted from the current configuration. We suppose that the requirement to add or to delete does not imply other requirements causing side effect phenomena to be managed. Thus, in order to evolve the application with a new requirement we augment the current selected configuration with a new feature implementing the new requirement. This leads to a new configuration that has not been anticipated at design time. Adding new requirements is more problematic than deleting requirements, thus we only discuss the first. Further, adding new behaviors seems to be appropriate for facing unforeseen situations.

In our approach we only evolve the current selected configuration whereas we do not consider how to augment the whole space of variants with the new requirement. We neither discuss how the addition of a new requirement to a configuration may affect the qualities attributes offered from the configuration.

The user may press a specific button within the application interface in order to communicate to the framework the variation of his/her needs. Then the user should specify the new requirement R_{New}, for example in natural language. The unforeseen evolution phase has to upgrade the running configuration with a new feature implementing the requirement R_{New}. We assume to have a search engine that given a requirement is able to return the set of features implementing it (exploration phase). Among them, we select the first feature $f_{New} = (R_{New}, I_{New}, C_{New})$ that is weakly consistent with the current running system configuration $G_F = (R_F, I_F, C_F)$ at the current context $\overrightarrow{c_{curr}}$:

$$(C_F \cup_C C_{new})[\overrightarrow{c_{curr}}/\overrightarrow{x}] \tag{6}$$

The integration phase creates the new configuration $G_F \cup_f f_{New}$ and the validation phase checks the weak consistency of the configuration at the current context state. The configuration is added to the configuration consistency table and since new resources may be required by the new feature it could be necessary to augment the context. Also the feature diagram is kept up-to-date by adding the incoming feature. We recall that in our approach, the integration of a new feature to the feature diagram only leads to a new configuration. We do not consider how to perform the integration of the new feature with all possible configurations since we only evolve the current configuration.

5.4 Working Example

Let us suppose that at the context state $C_{33} = (150, 400, 4096, 1)$, our framework completes a foreseen evolution for the fractal application. It puts in execution the configuration $G_{10}=\{f_{remGet}, f_{sockConn}\}$ which visualizes a precomputed fractal image after it has been downloaded from a remote server. The retrieved image complies to the TIFF image format. Because of unforeseen characteristics of the mobile device, the user cannot visualize the retrieved image. The device cannot decode TIFF images and therefore the fractal application has to be upgraded.

To this end, the user interacts with the framework to add a new requirement in the application. After accessing to the upgrading wizard, he/she specifies the new requirement in natural language:

$$R_{New} = \textit{The system shall visualize TIFF format images} \tag{7}$$

This requirement has not been foreseen at design time but arises only at run-time when the unforeseen device characteristics (context) make the fractal visualization impossible. Thus after the evolution process, we have to re-iterate the exploration, integration and validation phases at run-time in order to evolve the application with the feature (i.e. the software codec) to view TIFF format images. This will lead to a new configuration with same features of the current configuration plus the new feature.

The exploration phase queries the search engine in order to retrieve a feature which implements the new requirement, e.g. see Figure 11.

$I_{tiffViewer}$:

```
public class Viewer{ ...
 public RenderOp tiffViewer(Object stream){
  ParameterBlock params = new ParameterBlock();
  params.add(stream);
  TIFFDecodeParam decodeParam = new TIFFDecodeParam();
  RenderedOp image = JAI.create("tiff", params);
  return image;
}...}
```

$C_{tiffViewer}$: $cRate \geq 300 \wedge mem \geq 35$

Fig. 11. Example: new feature

The integration phase augments the feature diagram with the new feature as shown in Figure 12. An optional feature $f_{tiffViewer}$ is added to the diagram which only leads to a new configuration $G_{New} = \{f_{remGet}, f_{tiffViewer}, f_{sockConn}\}$.

For the validation phase we consider how the new context requirement affects the context requirements provided at design time. The new context requirement $C_{tiffViewer}$, that we consider for weak consistency, refers to the resources *cRate* and *mem* which have been already foreseen at design time; thus a context model extension is not required. To establish if the new configuration $G_{New} = G_{10} \cup_f f_{tiffViewer}$ is weakly consistent we evaluate the new context requirement jointly with the context requirement for G_{10}, i.e.:

$$C_{New} = cRate \geq 300 \wedge mem \geq 135 \wedge conn = 1$$

This predicate is true at the context state C_{33} since this state provides enough memory, cpu speed and an Internet connection. Only if the new predicate is false the framework does restart the evolution process by the exploration phase in order to consider other features. Finally, if the configuration is weakly consistent (the new predicate is true with the current context values), the validation phase adds the new configuration G_{New} to the consistency table as shown in Table 4 by checking the weak consistency property also for the other context states.

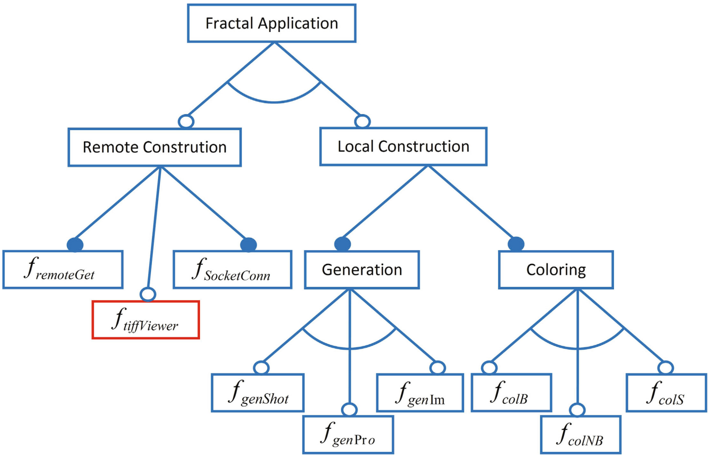


Fig. 12. Refined feature diagram

Even if it is not shown in the example, a new feature may also require new unforeseen context entities in its context requirements. Thus, it may be necessary to refine also the context model in order to consider the values for the new resources. As a consequence it would be also necessary to augment the consistency table with the new context states arising from the augmented context space.

Table 4. Refined configuration consistency table

$C(mem, cRate, sc, conn)/G_k$	G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8	G_9	G_{10}	G_{New}
$C_0 = (100, 200, 256, 0)$	0	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...
$C_{33} = (150, 400, 4096, 1)$	0	0	0	0	0	0	1	1	0	1	1
...	...	...	...	...	...	...	...	...	...	...	...
$C_{43} = (350, 200, 4096, 1)$	1	0	0	1	0	0	1	0	0	1	0
...	...	...	...	...	...	...	...	...	...	...	...
$C_{47} = (350, 600, 4096, 1)$	1	1	1	1	1	1	1	1	1	1	1

6 Evolution Framework Architecture

In Figure 13 is shown the architecture that can be implemented by any evolution framework in order to support the development and the execution of adaptive applications. This architecture implements the MAPE cycle as described in Section 1 and it supports our lifecycle process for context-aware adaptive systems. The application, configuration and feature blocks represent the basic components. They enable the definition of the application along with its variability. The context manager component is able to monitor the resources and to manage their definitions and values by accessing to the context model component. It performs the monitoring phase and it triggers the required evolution phases. The decision-making component maintains the context-based tables and the probabilistic automaton in order to support the decision-making mechanisms; it also supports the consistency checking phase and the ranking process for the configurations. A component for each kind of evolution is provided in the framework. As shown by the arrows, while the foreseen evolution accesses the decision-making component to select the most suitable configuration, the unforeseen evolution interacts with the user who specifies variation to the requirements. Finally the execution component enacts the system reconfiguration for both evolutions.

6.1 Framework Instantiation

We have instantiated the architecture in Figure 13 by exploiting current practices technologies available in the literature. We represent requirement R as Linear Time Temporal Logic expressions [39], whereas we represent the context requirements as predicates. We evaluate the context states in which a configuration is admissible by formalizing and solving a Constraint Satisfaction Problem (CSP) [35] by using the Java API available with the JaCoP tool[1]. Implementation artifacts are coded in Java, thus making it possible to verify the implementation components I with respect to the requirement R. To this end, our approach proposed in [23] defines a model checking phase which exploits the Java Path Finder tool[2].

Our framework supports the foreseen evolution by deciding which is the most suitable variant to execute whenever the current context state makes the running configuration not anymore admissible. To this end, the framework stops the execution of the running configuration and it puts the target variant in execution. Our earlier approach [36] describes how to select the most suitable variant based on the trade-off between user benefit and reconfiguration cost. The framework presented in this paper also supports the unforeseen evolution by exploiting a mechanism for the dynamic loading of Java classes. The user interacts with the application to specify a new requirement in a similar way as a programmer can add a new plug-in to the Eclipse or NetBeans IDE. If there is no configuration that can satisfy the augmented set of requirements, then the framework searches for a feature which implements the new requirement by interacting with a remote library of features. It creates a new configuration that contains the same set of features of the current configuration plus the new feature. The framework checks if the new set of features is free from interactions by evaluating context requirements. Once the framework has found such a new feature it gives as result the implementation for the new configuration. The framework supports the code replacement for the configuration and a mechanism for re-loading the new compiled classes (based on Javeleon[3]). Finally, the new configuration will be enacted trough its entry point method.

[1] `http://jacop.osolpro.com/`

[2] `http://babelfish.arc.nasa.gov/hg/jpf/jpf-core`

[3] `http://javeleon.org/`

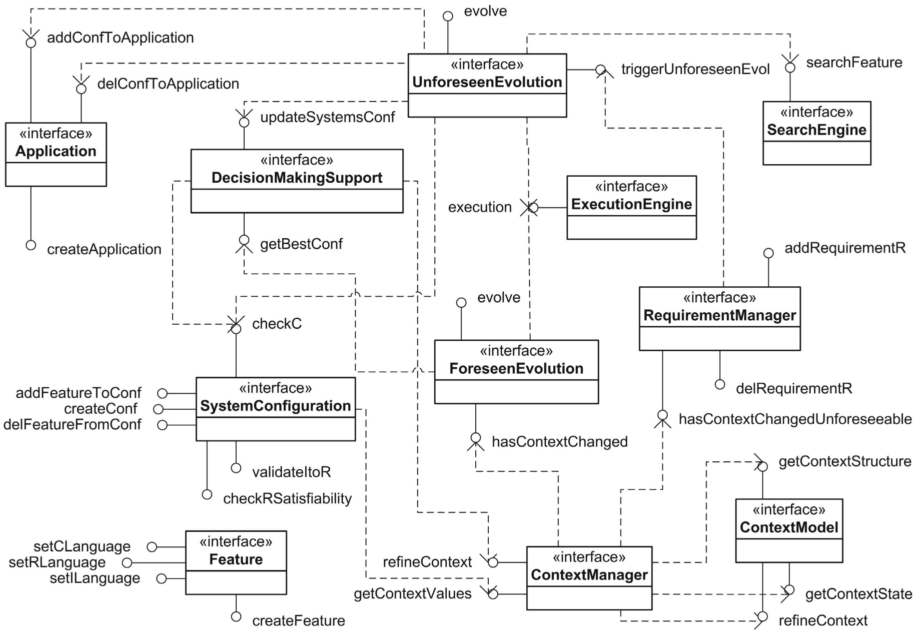


Fig. 13. Evolution framework architecture

7 Conclusion and Future Work

We have defined a generic model-centric software lifecycle process for context-aware adaptive systems. Our process supports concrete mechanisms to achieve consistent evolution both at design time and at run-time through a static and a dynamic decision-making procedure. We have proposed feature-oriented models to represent the system along with its variability and we have modeled context entities as the basis for the notion of weak consistent evolution.

We have defined a generic evolution framework in order to support the software process for adaptive systems. We have implemented a possible instance of the evolution framework applying current practice technologies.

As for future work, we will carry out extensive experimentations in order to evaluate advantages and disadvantages of adopting the framework to develop adaptive applications.

Acknowledgments. This work has been partially supported by the EU IST CONNECT (http://connect-forever.eu/) No 231167 of the FET - FP7 program and the EU IST CHOReOS (http://www.choreos.eu/) No 257178 of the FP7 program.

References

1. Alférez, M., Moreira, A., Kulesza, U., Araújo, J., Mateus, R., Amaral, V.: Detecting feature interactions in spl requirements analysis models. In: FOSD, pp. 117–123 (2009)
2. Ali, R., Dalpiaz, F., Giorgini, P.: A goal-based framework for contextual requirements modeling and analysis. Requir. Eng. 15(4), 439–458 (2010)
3. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Modeling dimensions of self-adaptive software systems. In: SEAMS, pp. 27–47 (2009)
4. Autili, M., Di Benedetto, P., Inverardi, P.: Context-Aware Adaptive Services: The PLASTIC Approach. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 124–139. Springer, Heidelberg (2009)
5. Autili, M., Benedetto, P.D., Inverardi, P.: Hybrid approach for resource-based comparison of adaptable java applications. Journal of Science of Computer Programming (SCP) - Special issue of BElgian-NEtherlands software eVOLution seminar (BENEVOL) on Software Evolution, Adaptability and Maintenance (2012)
6. Autili, M., Cortellessa, V., Ruscio, D.D., Inverardi, P., Pelliccione, P., Tivoli, M.: Eagle: engineering software in the ubiquitous globe by leveraging uncertainty. In: SIGSOFT FSE, pp. 488–491 (2011)
7. Baldauf, M., Dustdar, S., Rosenberg, F.: A survey on context-aware systems. IJAHUC 2(4), 263–277 (2007)
8. Berardinelli, L., Cortellessa, V., Di Marco, A.: Performance Modeling and Analysis of Context-Aware Mobile Software Systems. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 353–367. Springer, Heidelberg (2010)
9. Bisbal, J., Cheng, B.H.C.: Resource-based approach to feature interaction in adaptive software. In: WOSS, pp. 23–27 (2004)
10. Blair, G.S., Bencomo, N., France, R.B.: Models@ run.time. IEEE Computer 42(10), 22–27 (2009)
11. Brataas, G., Hallsteinsen, S.O., Rouvoy, R., Eliassen, F.: Scalability of decision models for dynamic product lines. In: SPLC (2), pp. 23–32 (2007)
12. 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.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009)
13. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.): Self-Adaptive Systems. LNCS, vol. 5525. Springer, Heidelberg (2009)
14. Classen, A., Heymans, P., Schobbens, P.-Y.: What's in a *Feature*: A Requirements Engineering Perspective. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 16–30. Springer, Heidelberg (2008)
15. Czarnecki, K., Eisenecker, U.W.: Generative programming: Methods, Tools and Applications. Addison-Wesley (2000)
16. Dorn, C., Dustdar, S.: Weighted fuzzy clustering for capability-driven service aggregation. In: SOCA, pp. 1–8 (2010)
17. Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B.R., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer 37(10), 46–54 (2004)
18. Ghezzi, C., Inverardi, P., Montangero, C.: Dynamically Evolvable Dependable Software: From Oxymoron to Reality. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 330–353. Springer, Heidelberg (2008)

19. Glinz, M.: On non-functional requirements. In: RE, pp. 21–26 (2007)
20. Gregersen, A.R., Jørgensen, B.N.: Dynamic update of java applications - balancing change flexibility vs programming transparency. Journal of Software Maintenance 21(2), 81–112 (2009)
21. Hong, J., Suh, E., Kim, S.-J.: Context-aware systems: A literature review and classification. Expert Syst. Appl. 36(4), 8509–8522 (2009)
22. Inverardi, P., Mori, M.: Feature oriented evolutions for context-aware adaptive systems. In: EVOL/IWPSE, pp. 93–97 (2010)
23. Inverardi, P., Mori, M.: Model checking requirements at run-time in adaptive systems. In: Proceedings of the 8th Workshop on Assurances for Self-adaptive Systems, ASAS 2011, pp. 5–9 (2011)
24. Inverardi, P., Mori, M.: Requirements models at run-time to support consistent system evolutions. In: Proceedings of the 2nd International Workshop on Requirements@Run.Time, pp. 1–8 (2011)
25. Inverardi, P., Tivoli, M.: The Future of Software: Adaptation and Dependability. In: De Lucia, A., Ferrucci, F. (eds.) ISSSE 2006-2008. LNCS, vol. 5413, pp. 1–31. Springer, Heidelberg (2009)
26. Jackson, M.: Problem Frames: Analyzing and structuring software development problems. Addison-Wesley Longman Publishing Co., Inc., Boston (2000)
27. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical report CMU/SEI-90-TR-21 SEI Carnegie Mellon University (1990)
28. Kapitsaki, G.M., Prezerakos, G.N., Tselikas, N.D., Venieris, I.S.: Context-aware service engineering: A survey. JSS 82(8) (2009)
29. Keck, D.O., Kühn, P.J.: The feature and service interaction problem in telecommunications systems. a survey. IEEE TSE 24(10), 779–796 (1998)
30. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Trans. Software Eng. 16(11), 1293–1306 (1990)
31. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: FOSE, Washington, DC, USA, pp. 259–268 (2007)
32. Mancinelli, F., Inverardi, P.: A resource model for adaptable applications. In: SEAMS, New York, NY, USA, pp. 9–15 (2006)
33. Mandelbrot, B.: The fractal geometry of nature. Freeman (1982)
34. Marco, A.D., Mascolo, C.: Performance analysis and prediction of physically mobile systems. In: WOSP, pp. 129–132 (2007)
35. Marriott, K., Stuckey, P.: Programming with Constraints: An introduction. MIT Press (1998)
36. Mori, M., Li, F., Dorn, C., Inverardi, P., Dustdar, S.: Leveraging State-Based User Preferences in Context-Aware Reconfigurations for Self-Adaptive Systems. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 286–301. Springer, Heidelberg (2011)
37. Osterweil, L.: Software processes are software too. In: ICSE, Los Alamitos, CA, USA, pp. 2–13 (1987)
38. Parra, C., Cleve, A., Blanc, X., Duchien, L.: Feature-Based Composition of Software Architectures. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 230–245. Springer, Heidelberg (2010)
39. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57 (1977)
40. Poladian, V., Garlan, D., Shaw, M., Satyanarayanan, M., Schmerl, B., Sousa, J.: Leveraging resource prediction for anticipatory dynamic configuration. In: SASO, Washington, DC, USA, pp. 214–223 (2007)

41. Pukall, M., Grebhahn, A., Schröter, R., Kästner, C., Cazzola, W., Götz, S.: Javadaptor: unrestricted dynamic software updates for java. In: ICSE, pp. 989–991 (2011)
42. Qureshi, N., Perini, A.: Requirements Engineering for Adaptive Service Based Applications. In: RE, pp. 108–111 (2010)
43. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. TAAS 4(2) (2009)
44. Sawyer, P., Bencomo, N., Whittle, J., Letier, E., Finkelstein, A.: Requirements-aware systems: A research agenda for re for self-adaptive systems. In: RE, pp. 95–103 (2010)
45. Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., Bontemps, Y.: Generic semantics of feature diagrams. Computer Networks 51(2), 456–479 (2007)
46. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. IEEE Trans. Software Eng. 33(12), 856–868 (2007)
47. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: ICSE, New York, NY, USA, pp. 371–380 (2006)
48. Zowghi, D., Gervasi, V.: The three cs of requirements: Consistency, completeness, and correctness. In: REFSQ (2002)

DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems

Norha M. Villegas[1,4], Gabriel Tamura[2,3,4], Hausi A. Müller[1], Laurence Duchien[2], and Rubby Casallas[3]

[1] University of Victoria, Victoria, Canada
{nvillega,hausi}@cs.uvic.ca
[2] INRIA - LIFL - University of Lille 1, Lille, France
{gabriel.tamura,laurence.duchien}@inria.fr
[3] University of Los Andes, Bogotá, Colombia
rcasalla@uniandes.edu.co
[4] Icesi University, Cali, Colombia

Abstract. Despite the valuable contributions on self-adaptation, most implemented approaches assume adaptation goals and monitoring infrastructures as non-mutable, thus constraining their applicability to systems whose context awareness is restricted to static monitors. Therefore, separation of concerns, dynamic monitoring, and runtime requirements variability are critical for satisfying system goals under highly changing environments. In this chapter we present DYNAMICO, a reference model for engineering adaptive software that helps guaranteeing the coherence of (i) adaptation mechanisms with respect to changes in adaptation goals; and (ii) monitoring mechanisms with respect to changes in both adaptation goals and adaptation mechanisms. DYNAMICO improves the engineering of self-adaptive systems by addressing (i) the management of adaptation properties and goals as control objectives; (ii) the separation of concerns among feedback loops required to address control objectives over time; and (iii) the management of dynamic context as an independent control function to preserve context-awareness in the adaptation mechanism.

1 Introduction

The necessity of a change of perspective in the engineering of software systems has been widely discussed during the last decade by several researchers and practitioners in different software application domains [1,2,3]. In particular, Truex *et al.* posited that software engineering has been based in part on an incorrect set of goals, from the assumption that software systems should support rigid and stable business structures and requirements, have low maintenance, and fully fulfill these requirements from the initial system delivery [4]. In contrast to this static and "stable" vision, they proposed a new set of goals based on permanent analysis, dynamic requirements negotiation and incomplete requirements

R. de Lemos et al. (Eds.): Self-Adaptive Systems, LNCS 7475, pp. 265–293, 2013.

specification. Their proposal is aligned with the vision of self-adaptive systems, where dynamic adaptation is necessary to ensure the continuous satisfaction of their functional requirements while preserving the agreed conditions on Quality of Service (QoS) levels. These QoS levels are usually represented in the form of Service Level Agreements (SLAs), and their enforcement mechanisms are based on contracts and policies, among others [5,6]. To achieve the continuous satisfaction of changing requirements, the development of this kind of systems requires adaptation mechanisms able to perform short-term adaptations on them, and manage their long-term evolution [7]. As part of this adaptation and evolution, system analysis must be performed at runtime, and its requirements satisfaction must be monitored and regulated by continuously adjusting or enhancing its behavior [8,3].

Although the feedback loop model of *control theory* has been used as a reference in many self-adaptive systems in different application domains, the visibility of the feedback loop as the crucial architectural element to govern software adaptation remains often hidden. In many cases, the managed application is intertwined with the adaptation mechanism, rendering it as hard to analyze, reuse, and manipulate [9,8,10]. In other cases, such as those following the multi-layer architectures (e.g., ACRA [11], FORMS [12] and Kramer and Magee's [13]), their designs assume a completely closed and controlled context where monitoring requirements are not subject to change, even though several feedback loops can be evidenced in them. However, for many systems it is not affordable to discard unexpected context changes and dynamic changes in adaptation goals and user requirements, such as SLA re-negotiation at runtime. In these cases, statically deployed context monitoring elements are not enough to cope with these levels of dynamics, which are implied by context unpredictability.

Hence, as context information requirements evolve over time, due not only to changes in the execution environment, but also to the evolution of the adaptive system and its requirements, monitoring infrastructures are also required to be self-adaptive. Furthermore, in these cases the adaptation of the monitoring infrastructure implies to update the context analyzer of the target system's adaptation mechanism. Therefore, these changes must be coordinated by an independent feedback loop, that is, the one that manages changing control objectives and adaptation goals at runtime, thus preserving context-awareness in the system evolution.

In this chapter we present DYNAMICO (Dynamic Adaptive, Monitoring and Control Objectives model), a reference model for engineering context-based self-adaptive software composed of three types of feedback loops. Each of these feedback loops manages each of the three levels of dynamics that we characterize for self-adaptation: (i) the control objectives feedback loop, (ii) the target system adaptation feedback loop, and (iii) the dynamic monitoring feedback loop. As a reference model (i.e., a standard decomposition of a known kind of problems into distinguishable parts, with functionalities and control/data flow that are well defined [14]), DYNAMICO calls self-adaptive system designers to be aware whether the objectives, the system, or the monitoring infrastructure must be adapted. In this sense, our reference model can be used to check if these dimensions are being considered in the designs. Moreover, it defines the elements and functionalities, as well as the control and data interactions to be implemented, not only among the feedback loop elements, but also among the three types of feedback loops. In addition, our characterization of the latter interactions allows our reference model to be applied partially, that is omitting any of its feedback loops, targeting self-adaptive systems where supporting changes in any of the three levels of dynamics is a crucial requirement.

In light of this, we argue that, in order to regulate the satisfaction of adaptation goals and managed application's requirements continuously, (i) each of the feedback loop elements and their interactions must be independently analyzable; and (ii) the monitoring elements must be able to process the different kinds of information that the varying context can produce appropriately. DYNAMICO was inspired by classical control theory and the autonomic element proposed by IBM researchers [15]. With this reference model we aim to contribute to the design of self-adaptive software by making its instances consider these aspects explicitly: (i) the achievement of adaptation goals and their usage as the reference control objectives; (ii) the separation of control concerns by decoupling the different feedback loops required to satisfy the reference objectives as context changes; and (iii) the specification of context management as an independent control function to preserve the contextual relevance with respect to internal and external context changes.

The remainder of this chapter is organized as follows. Section 2 describes an industrial-based application example that we use to explain our reference model and its application. In Sect. 3 we re-visit fundamental ideas and concepts that have shaped the engineering of self-adaptive software in the last years, and from which we distill our reference model. Section 4 presents our proposed reference model including the feedback loop interactions and their governance, as well as some variations that DYNAMICO admits. Finally, Sect(s). 5 and 6 discuss related work and conclude the chapter, respectively.

2 Application Example

This section presents a SOA governance application example based on an industrial case study we conducted in collaboration with the IBM Centre for Advanced Studies (CAS) Canada.[1] In this case study, self-adaptation mechanisms are exploited at runtime to manage service-level agreements (SLAs), and ensure quality of service (QoS) requirements in service-oriented systems [16]. In SOA and cloud-based systems QoS is highly affected by, and dependent on context information. On the one hand, SLAs may be violated at any time during system execution due to changes in the situation of relevant context entities such as computational infrastructure components (i.e., internal context), and users (i.e., external context). On the other hand, as businesses and users' requirements are evolving continuously, contracted QoS conditions (i.e., adaptation goals) may be

[1] http://www-927.ibm.com/ibm/cas/canada/research/index.shtml

frequently re-negotiated, thus affecting the effectiveness of monitoring and adaptation mechanisms. This application example is also based in one of our previous papers on *governance feedback loops* [17], where we applied our reference model to the implementation of a runtime governance infrastructure able to change monitoring strategies dynamically, as required by changes in adaptation goals and the adaptive system itself. The proposed self-adaptive governance infrastructure aims to ensure contracted conditions such as performance, reliability and resource consumption in SOA and cloud-based environments, where SLAs are constantly re-negotiated at runtime [18,17].

Software-as-a-Service (SaaS) is one of the business models in cloud computing environments. SaaS provides customers with several benefits such as maintenance and evolution supported by the cloud provider, high availability, pay-per-use, and low operational costs. Suppose an SaaS cloud provider, specialized in large scale e-commerce platforms, is interested in governing the efficiency of the service-oriented infrastructure with the goal of optimizing operational costs. Assume that to guarantee low operation costs and thus contracted conditions, performance governance has been initially defined as the adaptation goal. For this, a performance SLA defines a service level objective (SLO) to guarantee an efficiency measurement of at least 90% for a particular service (e.g., *ProcessingPurchaseOrder*). The metric associated to the SLO is the *time behavior metric (TB)* proposed by Lee *et al.* [19]. We used this metric as an efficiency measure based on the processing time of service interfaces. Let us assume that initially the *ProcessingPurchaseOrder* is composed only of one interface, thus we express the service efficiency as:

$$TB = \frac{ProcessingPurchaseOrder\ interface\ execution\ time}{total\ ProcessingPurchaseOrder\ service\ invocation\ time}\ . \qquad (1)$$

The denominator, *total ProcessingPurchaseOrder service invocation time*, represents the total time it takes for the service to respond after the corresponding request. The numerator, *ProcessingPurchaseOrder interface execution time*, indicates the time consumed for processing a given interface functionality. *ProcessingPurchaseOrder* is composed only of one task defined initially as one interface implementation, thus the numerator is the processing time required for executing that individual task (i.e., *total ProcessingPurchaseOrder service invocation time − waitingtime*). TB is in the range 0..1, where higher values indicate a better measure of performance in terms of time efficiency. Finally, suppose that an action guarantee, defined as part of the SLA, will trigger a self-optimizing feature that performs an on-line architectural reconfiguration to improve the system's efficiency and capacity.

2.1 The Need for Dynamic Context Monitoring

Using DYNAMICO, runtime SOA governance can be optimized by supporting adaptive monitoring strategies to address changes in monitoring requirements. Variations in monitoring requirements can be generated by changes in either the governance objectives (i.e, adaptation goals), the target system, the adaptation mechanism, or relevant context entities. The following two use cases illustrate the need for supporting dynamic monitoring, to preserve the context-awareness of the adaptation mechanism upon changes in the target system (i.e., changes in internal context entities) and adaptation goals.

Use Case 1: changes in internal context entities. Suppose that due to self-adaptation, the *ProcessingPurchaseOrder* service is replaced by a set of distributed services intended to enlarge the order processing capacity of the e-commerce platform. Consequently, the efficiency metric presented in (1) must be applied to every new service interface. With traditional static monitoring mechanisms, the governance of the performance SLA is compromised as the monitoring infrastructure was originally implemented to monitor the time efficiency of the *ProcessingPurchaseOrder* service interface only. The monitoring of the new interfaces is not supported without manually implementing the required sensors and monitors. This implies that every time monitoring conditions or the set of context entities to be monitored change, the monitoring instrumentation must be adjusted manually. Moreover, the effectiveness in performing these changes depends on the effectiveness in reporting them. Using our DYNAMICO, our SOA governance infrastructure is able to deal with changes in monitoring requirements at runtime. Once the new services for purchase order processing are deployed, adaptation mechanisms will trigger the adaptation of the monitoring strategy to monitor the new service interfaces. Our monitoring infrastructure exposes autonomous capabilities to configure and deploy new sensors and monitoring conditions at runtime. Implementation details regarding the dynamic capabilities of the implemented monitoring infrastructure are discussed in our MESOCA paper [17].

Use Case 2: changes in adaptation goals. Suppose now that the initial SLA is re-negotiated. A new service level objective (SLO) on throughput is added to the efficiency SLO defined originally as the contracted condition of the performance SLA (cf. (1)). The new throughput SLO defines two different throughput levels, depending on the applicable context situation, as summarized in Table 1 below.

The original monitoring infrastructure is implemented so that the initial performance SLA supports only the monitoring of the individual *ProcessingPurchaseOrder* interface. Once the SLA is re-negotiated, the adaptation mechanism is no longer effective as the new monitoring requirements imposed by the new throughput SLO are not supported. Dynamic changes in the monitoring infrastructure may occur at different levels. They may imply either the deployment of new sensors and new monitoring condition algorithms, or the modification of existing monitoring thresholds and conditions. In any case, without supporting changes in monitoring strategies at runtime, the adaptation mechanisms must be adjusted manually to ensure their relevance with respect to new adaptation goals. In this example, the new throughput SLO detailed in Table 1 will trigger the adaptation of the existing monitoring strategy. Two new sensors and corresponding monitoring conditions must be added. The first one is to monitor

Table 1. The throughput SLO is defined after the performance SLA has been renegotiated. The contracted conditions depend on different context situations that must be monitored.

Throughput SLO of the Performance SLA		
Throughput level	Monitoring Condition	Relevant Context Entities
Medium load	No. of likes on an offer $\leq$ 200,000	A special offer on a social network
Highest peak load	Is Black Friday or Christmas season?	Day of the year

the acceptance of a special offer placed on a social network integrated into the e-commerce platform. The second one is to keep track of the season. In both cases, the monitored information is used to anticipate the expected system load and thus modify the e-commerce platform capacity accordingly.

3 Design Drivers in the Engineering of Self-Adaptive Software

3.1 Feedback Loops

Feedback loops are the cornerstone of control theory, and as such, they provide the basis for automation in many fields of engineering and in particular for self-adaptation in computing and software engineering [20]. In this theory, the feedback loop or closed loop, as depicted in Fig. 1, is the *model* used to automate the control of dynamic systems. These control mechanisms are realized by comparing the *measured outputs* (A) of the target system behavior to the control objectives given as *reference inputs* (B), yielding the *control error* (C), and then adjusting the *controlling inputs* (D) accordingly for the target system to behave as defined by the reference input [9]. The measured output can also be affected by external *disturbances* (E), or even by the *noise* (F) caused by the system adaptation itself. *Transducers* (G) translate the signals coming from sensors, as required by the comparison element (H).

To keep objectives controlled in a target system, several strategies have been proposed. The three most common strategies are (i) the *regulatory* control, which ensures that the measured output is as close as possible to the reference input; (ii) the *disturbance rejection*, to control the effects of disturbances on the measured output; and (iii) the *optimization* control, which continuously seeks to obtain the best value of the measured output, as effectively as possible [9]. These strategies imply variations on the controller element but are realizable with the general structure of the block diagram. To compute the controlling signals, there are several possible mechanisms. In control theory, the representative mechanism is the *system transfer function,* a mathematical model built upon the physical properties and characteristics of the target system. Depending on these characteristics, the transfer function can be built, for instance, with proportional,

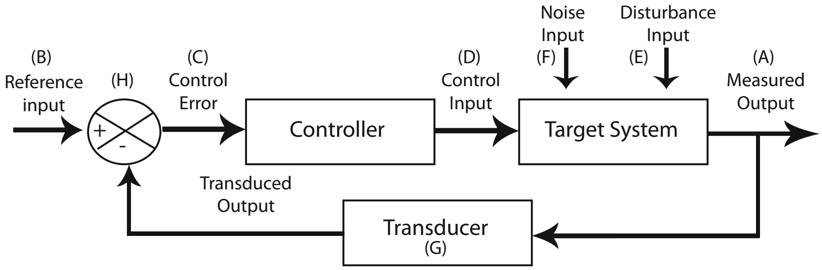


Fig. 1. Classical block diagram of a feedback control system [9]

derivative and integral (PID) terms. The parameters in a PID controller have special significance given that there exist precise and sophisticated methods for tuning their associated parameters.

Even though the application of control theory to industrial processes is well understood, its application to the control of software systems has at least two significant challenges: first, control theory is based on continuous mathematics, and second, it relies on measurements taken from, and actions performed into, physical, self-contained and self-performing artifacts (e.g., sensors, gauges and valves/actuators for temperature, pressure and other variables). As their associated variables are in the continuous-time domain, the use of continuous mathematics in this theory fits perfectly. In contrast, software systems are composed of intangible artifacts with discrete-time behavior and not always well characterized properties. Thus, direct sensing must be performed by CPU time-consuming software artifacts, and the adaptation mechanisms must reason on the target system's discrete-time output. Moreover, to exploit the possibilities of software adaptation fully, the output of the adaptation mechanism must be more structured than controlling signals to be transduced by electro-mechanical devices. This output may take the form, for example, of a plan of ordered actions to be instrumented by the software actuators on the target software components. Fortunately, there exists also the theory of linear discrete-time systems, which closely resembles the theory of linear continuous-time systems.

A reference model should not prescribe any particular software self-adaptation type of control. Instead, we propose DYNAMICO to characterize experimentally the effect that the controller actions produce in the observed behavior of the target system.

3.2 Visibility of Feedback Loops

The benefits of integrating feedback loop-based models into the engineering of self-adaptive software systems have been pointed out by several research papers [7,20,8,3]. Oreizy *et al.* define runtime adaptation in the form of two processes that exploit feedback loops to manage adaptation and system evolution, respectively. The evolution management process feedback loop is in charge of monitoring the consistency between architectural models and the actual system implementation. Whenever this consistency is no longer satisfied, the evolution management process feeds monitored information back to the adaptation management process feedback loop, which is in charge of reconfiguring the system's architecture [7]. Müller *et al.* outline on the benefits of specifying the feedback loops and their major components explicitly and independently. Furthermore, they articulate the usefulness of defining the interactions among the elements of a feedback loop explicitly, from analysis and design to implementation [20]. Giese *et al.* also argue for the decoupling of feedback loops in control-based reference architectures to address the satisfaction of quality attributes (control objectives), the management of the context complexity, and the interactions among multiple feedback loops and their elements [8]. Cheng *et al.* also emphasize the importance of making explicit not only the feedback loops, but also their elements and properties [3]. In fact, Müller *et al.* [20], as well as Kramer and Magee [13] attest that even though feedback loops have been recognized as fundamental design elements for self-adaptation, the related design documents and research publications usually hide the visibility of both the adaptation controller and the feedback loops. As a result, there currently exists no explicit methods for analysis, validation and verification useful to measure the effectiveness of adaptation mechanisms in software systems [21]. Based on these remarks, we aim in our reference model to increase the visibility of the feedback loop components by making them explicit entities of software architecture design and, thus, directly analyzable, assessable and comparable.

Valuable papers have been published making significant advances in the area. For instance, the feedback control architecture for adaptive systems proposed by Shaw decouples the elements of a feedback loop (i.e., comparison, plan correction, and effect correction), and identifies the importance of context relevance for the adaptation process [22,20]. In the same way, the autonomic manager (MAPE-K loop) presented in Fig. 2, and the autonomic computing reference architecture (ACRA) are important contributions of IBM that also make the feedback loops in autonomic systems explicit [11]. On the one hand, as explained

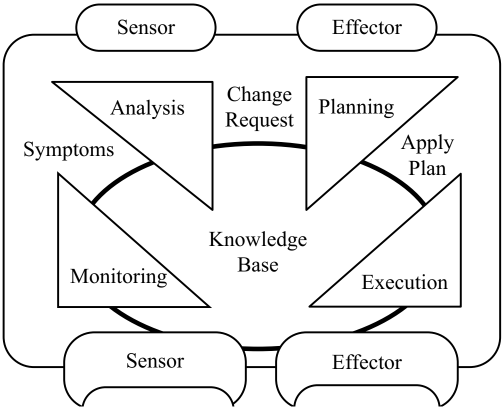


Fig. 2. The MAPE-K loop [15]

in [10], the autonomic manager is an implementation of the controller element in the generic control feedback loop depicted in Fig. 1. At the same time, the autonomic manager controls the managed element by implementing an intelligent control loop composed of the monitor, the analyzer, the planner, the executor, and the knowledge base elements. This knowledge base is an important element to share information along the loop. Moreover, it provides persistence for historical information and policies required to correlate complex situations. On the other hand, ACRA provides a reference architecture as a guide to organize and orchestrate an autonomic system. Autonomic systems based on ACRA are defined as a set of hierarchically structured building blocks composed of autonomic managers, knowledge sources and manageability endpoints (management interfaces).

Nonetheless, despite ACRA and the MAPE-K loop that have helped considerably improve the visibility of feedback loops, the internal components of each control loop, and the control loop itself, still remain hidden inside the autonomic manager. Certainly, the specification of the autonomic manager, provided in the IBM architectural blueprint for autonomic computing, characterizes the manager as *a component that implements an intelligent control loop* [11]. Moreover, even when the ACRA architecture drivers are clearly the feedback loops in the form of autonomic managers, their internal elements (i.e., the elements of the MAPE-K loop) are highly coupled. Therefore, even though the multiple feedback loops defined in an ACRA-based model can be distributed—for instance to improve the system scalability—this distribution is limited by the autonomic manager boundaries. Each autonomic manager implements the entire cycle to collect and aggregate information from the environment (monitor), to correlate the collected information and identify symptoms for supporting the adaptation decision making (analyzer), to plan the adaptation process (planner), and to perform the adaptation plan (executor).

The separation of concerns between the monitoring process, the adaptation controller, and the management of control objectives (adaptation goals) is still an open challenge. This challenge is crucial for governing the consistency between adaptation mechanisms and control objectives, while preserving the relevance of context monitoring of the adaptation mechanism. In light of this, we concluded that a loose-coupling schema is preferable to a tight-coupling one for the integration and communication among the feedback loop elements. However, we retain the idea of composing instances of feedback loops similarly as specified by the generic hierarchical structure described in ACRA. Finally, while the autonomic manager, as an implementation of the feedback loop, is the architecture driver for ACRA, our architecture drivers are the independent MAPE-K loop elements, their explicit interactions, and the separation of these elements in three main groups, as explained in the following section.

3.3 The Three Levels of Dynamics

We identify three levels of dynamics that must be controlled in the engineering of context-driven self-adaptive software systems: (i) the management of changing control objectives, (ii) the dynamic behavior of the adaptation mechanism controlling the target system, and (iii) the management of dynamic context information. Each of these levels of dynamics plays an important role in governing the dynamic nature of the other two levels. In the case of the first level, as business goals and corresponding control objectives that must drive the behavior of the adaptive system evolve continuously, context monitoring mechanisms (the third level of dynamics), and adaptation controllers (the second level) are required to change accordingly. Furthermore, the management of control objectives may be affected as a result of monitored observations at the third level of dynamics. For instance, whenever the system identifies that even though the adaptation mechanism is performing properly, control objectives may be reviewed to modify the adaptation and/or monitoring mechanism due to changes in context situations.

These three levels of dynamics may be clearly illustrated using the application example described in Sect. 2. The first level, the management of changing control objectives, corresponds to the software instrumentation required to identify changes in adaptation goals. In our example, these changes correspond to the re-negotiation of the performance SLA by adding a new throughput SLO to the initial efficiency SLO. The second level, the dynamic behavior of the adaptation mechanisms, refers to the capability of adaptation strategies to adapt according to changes in either adaptation goals, or context situations. In the application example used as illustration in this chapter, the adaptation mechanism does not expose dynamic behavior. That is, the adaptation strategy is always the same. The third level, the management of dynamic context information, refers to the instrumentation required to support changes in monitoring strategies at runtime. In the application example, the dynamic reconfiguration of the monitoring strategy is triggered by two different situations. In the first case, new sensors are deployed at runtime to monitor the new service interfaces that have been added with the new set of distributed services for processing purchase orders (cf. Sect. 2.1, Use Case 1). In the second case, new sensors and monitoring conditions are deployed dynamically due to changes in adaptation goals (i.e., the new throughput SLA). The negotiation of a new throughput SLO requires from the monitoring infrastructure to keep track of two new context entities, a special offer placed on a social network and the day of the year (cf. Table 1).

4 DYNAMICO: Our Reference Model

Bass *et al.* define a reference model in software engineering as a standard decomposition of a known kind of problems into clearly distinguishable parts [14]. Each of these parts has assigned a well defined functionality, and the data flow among these parts is explicitly specified. Reference models serve as starting points for software architecture and high-level design specifications.

Following this definition and based on the analysis of the seminal research presented in Sect. 3, we distill in our reference model the characteristics that have been commonly discussed and used in other representative research in the engineering of self-adaptive software. We started by considering the general feedback control loop block diagram presented in Fig. 1. In this diagram, the target system to be controlled, its controller and corresponding transducers are represented as rectangles. The elements for setting the reference input (set point) and perform the comparison against the system measurements are combined in a crossed circle. This block diagram reflects the relative simplicity of the "autonomous" but independent elements used in control engineering. This simplicity hides the very specific and natural electro-mechanical properties (e.g., resistance, capacitance, inductance) of these elements. In contrast, in the MAPE-K model the elements are interdependent and their functions are specified in a general way. Concerning the characteristics of the different control strategies, control theory takes advantage of exactly the particular complex properties of the matter that constitutes both the controller elements, as well as the system to be controlled. In the case of software artifacts, even though they lack the physical properties analyzed in control engineering, these artifacts are given particular properties of behavior by their particular design. Nonetheless, and because of this, it is practically impossible to generalize them.

Therefore, given the characteristics of software systems (i.e., the systems to be controlled), we find that the combination of a general specification for the common elements of both feedback-loops and MAPE-loops together with a loose coupling scheme, are the best options for DYNAMICO. Figure 3 captures these decisions, which represents the general component of our reference model. This diagram clearly results from the merging of the classical feedback-loop and the MAPE loop model (cf. Fig. 1 and Fig. 2 respectively).

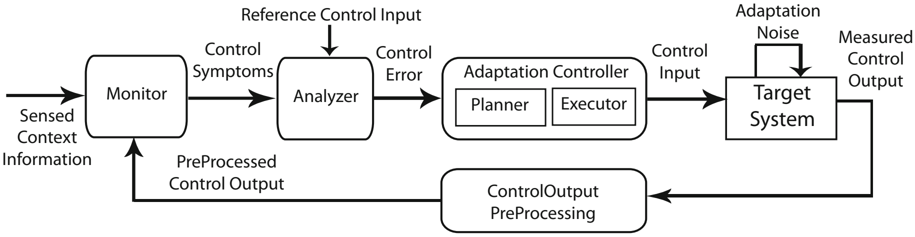


Fig. 3. General components of DYNAMICO. Feedback control block diagram with explicit functional elements and corresponding interactions to control dynamic adaptation in a software system.

4.1 Addressing Separation of Concerns

Analyzing Fig. 3 from both the control theory and software architecture perspective, for a software system (*target system*) to become effectively context-driven self-adaptive, it should incorporate at least three subsystems: (i) a *control objectives manager*, (ii) an *adaptation controller mechanism*, and (iii) a *context manager* or *monitoring infrastructure controller mechanism*. This design separates the concerns with respect to the three levels of dynamics we have proposed as design drivers for the engineering of context-driven self-adaptive systems: (a) the regulation of the target system's functional and non-functional requirements satisfaction; (b) the continuous accomplishment of adaptation goals and the preservation of the target system's properties under changing conditions of execution; and (c) the relevance of the context monitoring infrastructure according to the varying execution environment (dynamic context monitoring). This separation of concerns leads us to abstract the block diagram presented in Fig. 3 into the block diagram presented in Fig. 4. In this diagram, which constitutes our reference model as such, each of the three feedback loops, the *control objectives feedback loop* (CO-FL), the *adaptation feedback loop* (A-FL), and the *monitoring feedback loop* (M-FL), is an instance of the model depicted in Fig. 3.

The identification of these subsystems as independent feedback loops allows us to independently analyze, design, implement, and assess the instrumentation required to address the complexity of changing requirements at each of the three levels of dynamics. In this way, and depending on the nature of the adaptive system, this instrumentation can be easily temporal and spatial distributed and maintained. In addition, the entire software system would be less affected by

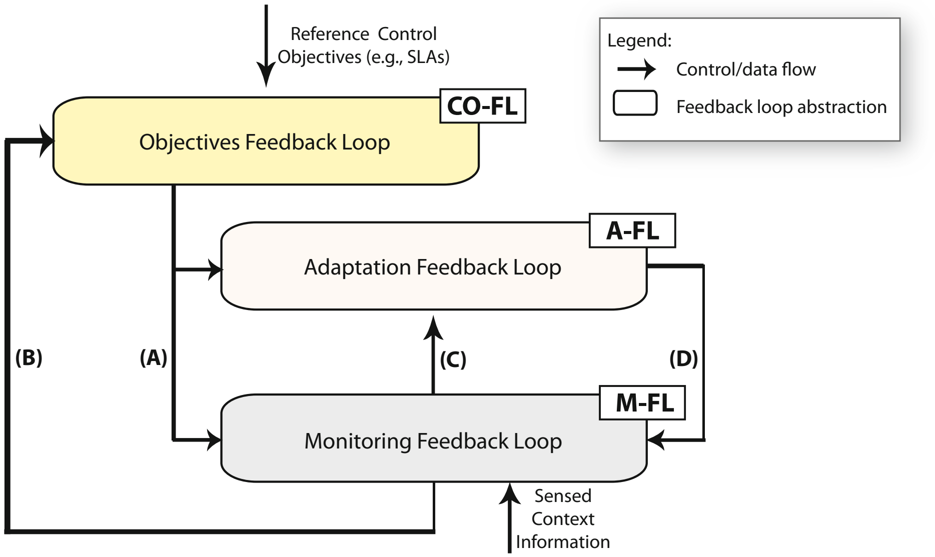


Fig. 4. The three levels of dynamics that must be controlled in context-driven self-adaptive software systems. The control objectives feedback loop, (CO-FL), controls changes in adaptation goals and monitoring requirements to ensure their fulfillment. The adaptation feedback loop, (A-FL), controls the adaptive behavior of the target system and the adaptation mechanism, according to control objectives and taking into account monitored context events. The dynamic monitoring feedback loop, (M-FL), manages context information for preserving context relevance of the adaptation mechanism. Labels (A), (B), (C) and (D) highlight the control/data flow among the feedback loops, which would require the implementation of the appropriate method interfaces.

the computational effort of each of the three subsystems. The separation of concerns made explicit by the DYNAMICO model is particularly crucial for cases such as the cloud-based e-commerce platform presented in our application example. In this example, the automatic reconfiguration of the monitoring strategy would not be feasible without having the context manager as an independent implementation of the adaptation mechanism. In the same way, the explicit control of changes in SLAs requires separate instrumentation. In the case where a dynamic adaptation mechanism is necessary, having a self-contained adaptation strategy (i.e., planner and executor) will contribute to the preservation of desired properties. The chapter "On Patterns for Decentralized Control in Self-Adaptive Systems", by Weyns *et al.* in this book, presents useful architectural patterns, that can be combined with our reference model, for implementing distributed and decentralized feedback loops in self-adaptive software systems.

By applying the separation of concerns introduced in our reference model, it is possible to support three different types of adaptation, depending on the different interactions implemented among the feedback loops: *preventive, corrective* and *predictive.* In preventive adaptation, the dynamic monitoring feedback loop notifies the adaptation feedback loop about context events (*context symptoms*) that, even when they are causing no effects yet in the target system behavior, they eventually will. This is the case of the monitoring condition that evaluates the number of *likes* of an offer placed on a social network. As the offer becomes popular, that is, the number of likes is close to 200,000, a predictive adaptation process can be started to take the system to its medium-load capacity (cf. Table 1 in Sect. 2). Consequently, even though the adaptation subsystem has not detected any disturbances yet for triggering adaptation in the target system, based on this context information, it can minimize the risks of the goal satisfaction to be violated by performing a system adaptation in advance.

Corrective is the usual type of adaptation that takes place when monitoring mechanisms supporting the adaptation feedback loop detect adaptation goals are no longer satisfied. In our application example this can occur when the monitoring feedback loop identifies an SLO violation in either the efficiency of the *ProcessingPurchaseOrder* service(s), or the expected minimum number of purchase orders processed per unit of time (cf. Table 1). Any of these situations requires from the adaptation controller to perform another, perhaps more aggressive, system reconfiguration, or to apply restrictive mechanisms of use to prevent the system from collapsing before a new adaptation is performed.

Predictive adaptation takes advantage of both, historical information to anticipate risks of goal violation, as well as the identification of plausible symptoms that provide evidence to necessitate adaptation eventually. These symptoms may be presented in the form of patterns of correlated events that potentially become significant advice for adaptation. An example of this latter case in our application scenario is the detection of a low but constant degradation of the *ProcessingPurchaseOrder* service efficiency around significant dates, but without reaching the critical levels that trigger corrective adaptation. Using this historical information, the dynamic monitoring feedback loop can trigger an alert event to indicate or notify the operators that the negotiated performance SLA should be reviewed to keep the system operation in a safe state.

Finally, it is worth noting that in Fig. 4 despite this separation of concerns, the control objectives feedback loop (i.e., CO-FL in the figure), the adaptation feedback loop (i.e., A-FL, including the target system), and the dynamic monitoring feedback loop (i.e., M-FL) together, also constitute a feedback loop. Figure 5 presents the detailed view of the reference model where each level of dynamics is designed as an instance of the general feedback loop with explicit components required for controlling the self-adaptation in software systems.

4.2 The Control Objectives Feedback Loop (CO-FL)

In DYNAMICO, the regulation of requirements satisfaction and the preservation of adaptation properties are objectives controlled through the collaboration of the A-FL and the M-FL. We define requirements and adaptation properties as system variables to be controlled. Throughout the chapter, we refer to these variables as *control objectives* and *adaptation goals* interchangeably. These requirements can be functional and non-functional, and the target system must satisfy them, depending for this on the adaptive capabilities of the overall system. Adaptation properties refer to the properties that are inherent in self-adaptive software, and thus, all adaptation mechanisms should expose these properties [21]. As mentioned in Sect. 3.3, these control objectives are subject to change by user-level (re)negotiations at runtime and therefore must be addressed in a consistent and synchronized way by the adaptation mechanism and the context manager. There may be several causes for these changes. In a first case, service level agreements with dependencies on context situations can imply changes in control objectives at runtime. In our application example, this is the case of the throughput SLO (cf. Table 1). This SLO defines two different thresholds. The medium load threshold is applicable to those cases where special product offers are placed online (e.g., on a social network integrated to the business e-commerce platform). After placing the offer, it must be monitored to apply preventive adaptation with the goal of adjusting the system capacity according to the popularity of the offer. Popular offers are expected to affect the e-commerce platform load considerably. Similarly, time context must be monitored to keep track of the shopping seasons to apply preventive adaptation to guarantee the system operation when the system load reaches its highest point (cf. Table 1). In another case, when the system is in execution, the initial SLA conditions can be re-negotiated. An instance of this case occurs in the second use case of our application example. After the contracted services for the e-commerce platform have been in production, a new throughput SLA is added to the efficiency SLO agreed initially. Both the throughput and efficiency SLOs are managed explicitly as the control objectives for the adaptive system. Thus, both reference inputs, the A-FL reference control input, and the M-FL reference context input, should be derived automatically from changes in control

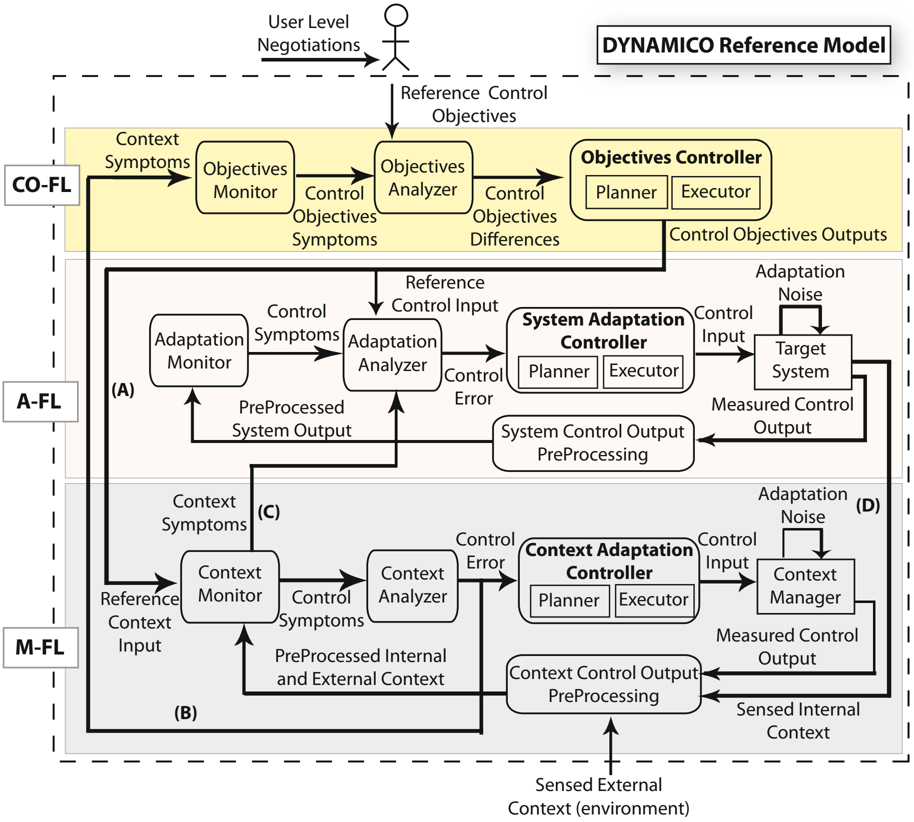


Fig. 5. Our DYNAMICO reference model with a detailed view of the controllers for the three levels of dynamics presented in Fig. 4 realized as the control objectives feedback loop (CO-FL), the adaptation feedback loop (A-FL), and the monitoring feedback loop (M-FL), respectively

objectives and fed into the corresponding feedback loops, as illustrated by interaction (A) in Fig. 5. All of these changes between SLOs and SLAs, which are treated as changes in reference inputs, are governed by the CO-FL.

Nonetheless, this explicit management of control reference inputs has two important implications: (i) it is required to model and express the corresponding properties quantitatively in terms of quality attributes, and (ii) it is necessary to have a mechanism to measure and update these reference inputs at runtime whenever they change. Concerning these two implications, we proposed a comprehensive evaluation framework composed of a set of adaptation properties and adaptation goals, and corresponding quality attributes [21]. This catalog is useful for the assessment of self-adaptive software based on the accomplishment of control objectives and suitability of adaptation mechanisms. Concerning the second implication, the dynamic adaptation of control reference inputs (control objectives) is addressed by *closing* the CO-FL. In the context of the main loop (the one composed of the three feedback loops), the A-FL receives symptoms from the M-FL through interaction (C), which in turn adapts its behavior according to changes in control objectives to guarantee monitoring relevance along the adaptation process. Under more dynamic scenarios, the A-FL controller may be required also to change its adaptation strategy according to changes in control objectives. Furthermore, as an important concern in service provision is the fulfillment of SLAs as specified in contracts, a plausible way to express and manage these reference goals quantitatively is through *contract management* and its explicit modeling.

4.3 The Adaptation Feedback Loop (A-FL)

The adaptation feedback loop, A-FL, serves as a guarantor for regulating the target system's requirements satisfaction and preserving the adaptation properties. Recalling our application example, the efficiency and throughput SLOs represent system's requirements. Due to the changing nature of SLAs and context situations, the satisfaction of these requirements depends on the adaptive capabilities of the e-commerce platform. Among the adaptation properties applicable to the adaptation mechanism of the application example are settling time, small overshoot, stability, and reconfiguration termination. In particular, settling time, the time it takes for the adaptation mechanism to complete the e-commerce platform reconfiguration, is crucial to guarantee the contracted conditions. Our SEAMS 2011 paper provides a comprehensive catalog of adaptation properties and corresponding quality attributes and metrics [21].

A-FL follows the separation of concerns criteria of the previous section. In turn, these criteria conform to the general protocol of control theory, which relies on quantitative expressions to measure the error in the controlled system variables, and respective reference control inputs for these variables. The A-FL gathers these measurements continuously from the target system through context monitors. These monitors notify control symptoms for adaptation to the A-FL analyzer, which determines whether a system adaptation is required (cf. analyzer in Fig. 3). The simplest case for this occurs when the measured variables under control, compared to their corresponding reference control inputs, indicate that some control objective is no longer satisfied. Whenever it is relevant, the A-FL analyzer notifies this fact with the corresponding information to the system adaptation controller. With this information, the planner element selects a strategy to adapt the system for it to re-establish the fulfillment of the violated control objective. A possible result of this strategy is to compute and send a list of system architecture reconfiguration actions to the executor (e.g., a set of distributed services to replace the original *ProcessingPurchaseOrder* service). The executor translates these actions to the specific runtime platform and executes them in the target system, thus closing the main control loop. DYNAMICO and its A-FL can take advantage of any strategy to perform the target system adaptation.

4.4 The Monitoring Feedback Loop (M-FL)

The role of the monitoring feedback loop, M-FL, as an independent feedback control loop is crucial for addressing the dynamic nature of context information. In a *context-based self-adaptive system*, a context manager must be able to make decisions based on past, current and foreseeable future states of context. It must analyze context symptoms and facts to support the system adaptation and the management of control objectives, as explained in Sect. 4.2. Moreover, the monitoring mechanism must adapt itself to support new context management requirements as the common control objectives are re-negotiated, or the adaptive system evolves. For instance, the context manager for the application example must be able to deploy new context management instrumentation. In the first use case, the deployment of the new set of distributed services, caused by the adaptation of the e-commerce platform, will trigger the deployment of a new set of time behavior sensors to keep track of the new service interfaces (cf. Sect. 2.1, Use Case 1). In the second use case, the re-negotiation of the performance SLA (cf. Sect. 2.1, Use Case 2) will trigger the deployment of the monitoring infrastructure required to keep track of two new types of context information, the shopping season (i.e., time context according to our *Smarter-Context* taxonomy [23]), and the special product offer (i.e., artificial context).

The M-FL in Fig. 5 represents a context manager that supports dynamic monitoring. The reference context inputs correspond to the reference context management objectives derived from the CO-FL reference control objectives. Context monitors are in charge of gathering primary context information from the internal and external environment, and the correlation of this information to infer either, context symptoms that can affect the target system adaptation process (provided to the A-FL through interaction (C) in Fig. 5), or control symptoms to decide about the context manager adaptation. This information is pre-processed by the context control output preprocessing element to generate numeric observables from physical and logical sensors, and producing comparable measures by performing basic transformations on them.

The context analyzer performs the context handling process required for the context adaptation controller to decide about adapting the monitoring strategy, and for the CO-FL to decide about changing the system objectives (interaction (B)), as demanded by the current state of the environment and the self-adaptive system requirements. The change of control objectives can be performed fully- or semi-automatically, depending on whether it is necessary to re-negotiate the contracts, and consequently, for the user to intervene (cf. Sect. 4.2). The context adaptation controller is responsible for defining and executing the adaptation plan for the context manager, according to its adaptation strategy.

Finally, the measured control output and the target system's internal context are used to ensure the context manager goals, thus supporting the system adaptation process and the management of the system control objectives.

To explicitly manage the relationship between control objectives and monitoring requirements in our case study, we proposed context-driven SLAs [17]. A *context-driven SLA* is an extension of a traditional SLA where context requirements are explicitly mapped to SLOs. In this way, changes in SLOs generated at runtime will include changes in the context management strategy specified with the original SLA. Context-driven SLAs are implemented as *contextual RDF graphs* based on the *SmarterContext ontology*. Both contextual RDF graphs and SmarterContext are results of our research on dynamic context management for context-aware self-adaptive software systems [23,17].

From the reference context inputs stated with the SLA, it is possible to generate context models that represent the environmental information relevant for the adaptation process. In our application example, context models are RDF graphs that represent a composition of relevant context entities, context sensors, and monitoring conditions. Whenever new SLAs are defined or existing ones are re-negotiated, the RDF representation of the monitoring strategy for the corresponding SLOs must be updated accordingly. The contextual RDF graph representing the new monitoring requirements is processed by our M-FL analyzer. Then, the planner element of the M-FL generates the adaptation plan that will modify the monitoring strategy by deploying new, or modifying existing sensors and monitoring conditions. The generation of these context adaptation plans at runtime is based on semantic Web inference rules defined as part of our SmarterContext ontology. Further details on the instrumentation of dynamic monitoring strategies for our case study are available in [17].

4.5 Feedback Loop Interactions

In DYNAMICO, not only are the three described feedback control loops well separated, but also the elements within each feedback loop. However, even though control loops are designed independently of each other, they must operate cooperatively to achieve the overall system objectives.

As depicted in Figs. 4 and 5, to regulate the satisfaction of the control objectives, DYNAMICO specifies four interactions among its three feedback loops. These interactions are labeled (A), (B), (C) and (D) in Fig. 5. We classify interactions (A) and (B) as *indirect interactions* because they are realized through the CO-FL, whereas interactions (C) and (D) as *direct interactions* due to their direct connections between the M-FL and the A-FL.

Interaction (A) provides the reference context input (i.e., context manager requirements) for the context manager (M-FL) to (i) maintain its relevance with respect to the actual context situation and contracted conditions; and (ii) decide on context management strategies. In the application example, reference context inputs correspond to the context management requirements defined as part of the SLA in the form of contextual RDF graphs [17].

Interaction (B) enables the control objectives manager (CO-FL) to decide about the changes in the control objectives, whenever the M-FL detects that, given the current context, the current set of control objectives should be adjusted or re-negotiated dynamically. Common control objectives are crucial for governing the interactions between the A-FL and the M-FL. We specify common control objectives in the form of contracts, machine readable SLAs as contextual RDF-graphs to infer both, adaptation and context monitoring objectives

[24,6,17]. Thus, a context management infrastructure (i.e., M-FL) must be able to infer, from contracts and common control objectives, the context management reference inputs, as well as the required monitoring strategies.

Interaction (C) is triggered by context symptoms that are identified and sent from the M-FL context monitor to the A-FL analyzer. These context symptoms, which can be manifested as groups of events presented with different characteristics, are important for decision making in the A-FL. The communication mechanism and the information associated with these symptoms depend on the type of adaptation the system is supporting (i.e., preventive, corrective or predictive). For example, for a predictive adaptation, the M-FL could trigger symptomatic events in advance about whether or not to perform a future adaptation. For a preventive adaptation, the M-FL also sends symptoms, but the adaptation is performed immediately. In contrast, for corrective adaptation, symptoms are either, pushed by the M-FL or pulled by the A-FL depending on who recognizes the need for adaptation —the context manager or the adaptation controller.

Interaction (D) represents the flow of internal context sensed by the M-FL from the adaptive system. Monitoring of internal context information is necessary to assess the system consistency after an adaptation. Moreover, by analyzing internal context information that characterizes the current state of system properties, the M-FL could provide useful information to understand the relationship between context symptoms, achievement of system goals, and the preservation of adaptation properties [21].

4.6 Governing and Controlling Feedback Loop Interactions

According to our reference model, an adaptive system is defined as a collection of cooperating feedback loops that ensure the achievement of the system objectives under changing context conditions. However, DYNAMICO can be combined with other models for adaptive systems. In particular, the IBM architectural blueprint provides the ACRA model to orchestrate control loops hierarchically for autonomic systems [11,15]. Combined with this model, DYNAMICO supports the distribution of functions in a more fine-grained level, that is, at the feedback-loop elements level. More extensive use of knowledge bases, as the ones proposed for the MAPE-K loop, should also facilitate interactions among control loops. Such knowledge bases store historical information such as symptoms, as well as internal and external context facts required by the analyzers in any of the three types of control loop. Moreover, these persistence mechanisms help to fine-tune contracts and policies to achieve the control objectives, and to develop machine-learning based adaptation mechanisms [25]. It is worth noting that having common control objectives enable the three control loops to reason consistently about the system goals, and to determine the coordinated control actions on each of them.

Having common control objectives is important to govern the interactions among the feedback loops. Figure 6 illustrates DYNAMICO abstracted as a control objectives feedback loop. The A-FL (adaptation mechanism), the M-FL (context manager), and the core controlled target system are abstracted as a whole managed *(super target) system*. This managed system is governed by the CO-FL according to changes in contracted conditions. Reference control objectives (i.e., contracts) are fed into the system through direct user intervention. Changes in these objectives can result from re-negotiations or from context symptoms received through interaction (B) (cf. Fig. 4). According to Fig. 6, whenever the objectives change as a result of symptoms received from the context manager, the control objectives monitor perceives these symptoms as symptoms of changes in the current set of control objectives. Then, the CO-FL analyzer makes decisions on the necessity of producing a new set of reference control inputs. If applicable, the CO-FL controller produces a new set of reference control inputs and reference context inputs to be sent to the adaptation mechanism and the context manager respectively. The measured control objectives feed the system back with information about the achievement of the system control goals. Finally, if the control objectives change as a result of a re-negotiation, the user is responsible for providing the control objectives analyzer with the new SLAs, and their corresponding SLOs and context monitoring requirements.

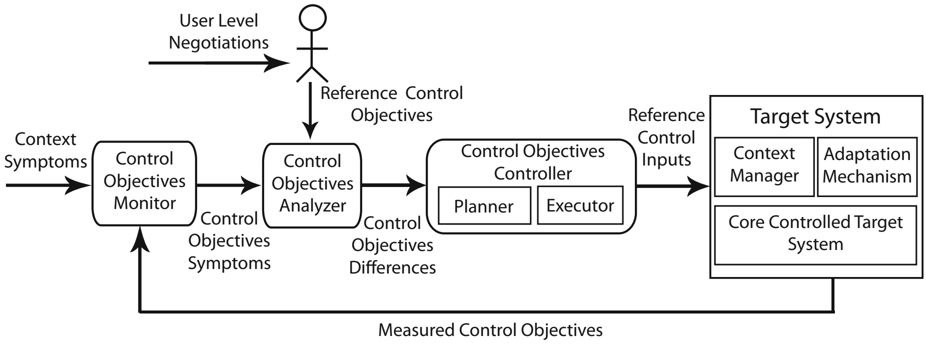


Fig. 6. DYNAMICO abstracted as a feedback loop for governing the dynamic change of the system control objectives

4.7 Possible DYNAMICO Variations

To deal with out-of-kilter environmental behaviors or perturbations, control community has developed several variations to modify the control function, such as the Model Reference Adaptive Control (MRAC) and the Model Identification Adaptive Control (MIAC) mechanisms [26,27]. The main difference between MRAC and MIAC is how the reference model is defined—in MIAC directly inferred from the running process, whereas in MRAC pre-computed using a mathematical model.

These variations are also applicable to DYNAMICO. Both variations can be realized, for instance, using a rule-based or policy-based reconfiguration approach in the planner element of the system adaptation controller, as illustrated in Fig. 7. In this figure, the CO-FL is represented by the control objectives manager. The adjustment mechanism detects, through the measured control output, whether the target system is facing an out-of-kilter environmental perturbation (e.g., an unusual high number of on-line shoppers during the Black Friday season), or the adaptation strategy is far from being effective. If this is the case, it modifies either the system adaptation planner (in the adaptation mechanism), or the context adaptation planner (in the context manager), depending on the situation. In our application example, the adaptation planner can be adjusted by replacing the reconfiguration rules in the rule-based subsystem using the *controller parameters*. Similarly, the context manager's planner could be modified by replacing the semantic Web rules, defined as part of the SmarterContext ontology, to be used to infer changes in monitoring strategies.

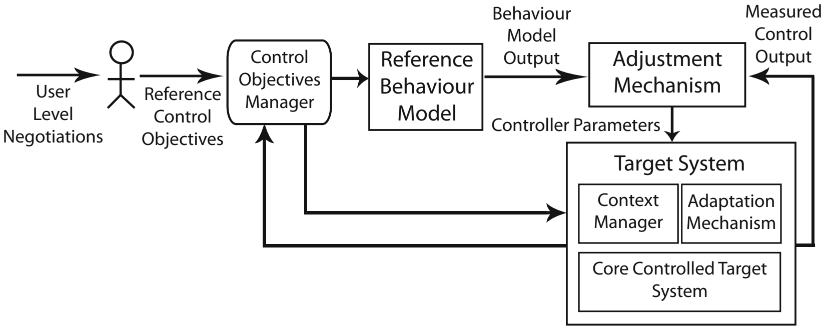


Fig. 7. Reference model variation for supporting adaptive feedback control loops with reference behavior models. The control objectives manager feedback loop is abstracted.

5 Discussion of Related Work

Different research communities, related to dynamic software systems, have proposed several examples of the application of feedback loops to concrete implementations of this type of systems. However, even though in most cases the existence of the feedback loop is evident, their designs lack separation of concerns among the multiple feedback loops required to orchestrate the three levels of dynamics introduced by our reference model (i.e., CO-FL, A-FL, and M-FL). Moreover, the explicit treatment of the interactions among these three levels is not generally addressed by existing implementations. Our reference model is general enough for being applied to different context-driven adaptive systems in many different application domains, where supporting changes in the three levels of self-adaptation dynamics is a crucial requirement.

In this section we discuss how DYNAMICO can be used to optimize context relevance in existing implementations of self-adaptive approaches, as well as the way different models for self-adaptation address the key drivers addressed by our reference model.

5.1 Optimizing Existing Implementations

A first example of concrete implementations is Rainbow, the adaptive framework for implementing self-healing software systems developed by Garlan *et al.* [28]. Rainbow's architecture maps directly to the feedback control architecture proposed by Shaw [22,20]. Our contribution complements Garlan's and Shaw's approaches by making explicit not only the feedback loops, but also their internal components, the interactions among them, as well as the separation of concerns at the three levels of dynamics proposed by our reference model.

A second interesting instance from a different application domain is the context-aware dynamic software product line proposed by Parra *et al.* [29]. They proposed the introduction of context-aware assets that are dynamically incorporated into the product line, depending on context changes. Although their architecture identifies the main feedback loop elements—a context manager (monitor), a decision maker (analyzer and planner), a runtime platform (executor) and a knowledge base—DYNAMICO can be used to improve their architecture by introducing a context monitoring infrastructure governed by an independent feedback loop, and coordinating the respective feedback loop interactions.

Yet another instance from the autonomic computing community is the real-time adaptive control approach for autonomic computing environments proposed by Solomon *et al.* [30]. Their system aims to control the computing infrastructure through a mathematical description of the time variation on the number of users in the system. Based on this function, the system modifies the control structure of the autonomic computing infrastructure by replacing its controller with one that matches the variation of the number of users on given time intervals. Furthermore, their adaptive control is based on a multi-layer architecture similar to ACRA, where the two upper layers correspond, respectively, to the autonomic system adaptation and the autonomic system layers, and the lowest layer corresponds to the managed infrastructure. The autonomic system adaptation layer adapts the autonomic system layer whenever the management objectives are not achieved. In this particular case, DYNAMICO is valuable for addressing the separation of concerns within the adaptation and autonomic management layers, as well as to guarantee the contextual relevance of monitoring mechanisms according to changes in the management objectives.

In the self-organizing systems community, Caprarescu and Petcu proposed a decentralized autonomic manager composed of many independent lightweight feedback loops implemented as agents, where each agent is an implementation of a MAPE-K loop [31]. Control objectives in this approach are specified as policies. Moreover, each feedback loop agent uses just one policy that is shared among all the agents organized in the same group. At the architectural level, this approach is based on the three-layer model proposed by Kramer and Magee [13], which was in turn inspired by the three-layer architectures proposed by the artificial intelligence and robotics community [32]. The system performs its adaptation based on a process of three phases. The first one separates agents into groups according to policies (i.e., self-organization phase); the second one ensures that only one agent can execute changes at a specific time (i.e., management phase); and the third one keeps the policies of the feedback loop up to date (i.e., policy update phase). Feedback loops adapt the system by modifying their parameters, adding new components or reconnecting components. The application of our reference model to this self-organizing system would help tackle the high degree of coupling among the components of each feedback loop, thus making the system components replaceable, reusable and distributable. An instance of the application of our reference model to this particular domain is the self-healing distributed scheduling platform presented by Frîncu *et al.* [33].

5.2 Comparing DYNAMICO to Other Self-Adaption Models

With DYNAMICO we intend to provide software engineers with a simple, but useful guide to (i) identify the minimum components required for implementing highly dynamic adaptive systems (i.e., facing highly changing contexts); and (ii) realize and control effectively the interactions among these components at runtime. Thus, our reference model aims to support software engineers in the implementation of dynamic mechanisms, by calling their attention to the necessity of reasoning about changes at the three levels of dynamics introduced by DYNAMICO. Moreover, our model constitutes a guide to analyze the effect of these changes in (a) the accomplishment of control objectives, (b) adaptation mechanisms, and (c) context relevance along the system evolution. From the perspective of this research, highly dynamic adaptive systems are adaptive systems where changes in control objectives (adaptation goals) are supported at runtime. As a result, adaptation and monitoring mechanisms are capable of adjusting themselves, at runtime, accordingly. To address dynamics, feedback loops, their visibility, and separation of concerns among them and their components constitute key runtime drivers in DYNAMICO.

Several contributions have recognized the importance of these drivers in the engineering of self-adaptive software. Feedback loop models from control theory address separation of concerns by decoupling controllers from target systems. From the perspective of adaptive software, this corresponds to a separation of concerns between adaptation mechanisms and managed systems [9]. The autonomic manager, as defined by IBM in its autonomic computing vision, goes further by increasing the visibility of the components that define a controller in the form of the MAPE-K loop. Moreover, the autonomic manager identifies the knowledge base as an important element for implementing intra-loop communication and data persistence mechanisms. Similarly to feedback loops, the autonomic manager addresses separation of concerns by implementing sensors and effectors as a level of indirection between the adaptation mechanism and the managed element [15]. A more recent reference model is FORMS, defined by Weyns *et al.* [12]. FORMS provides a meta-model based on the MAPE-K model, and combines it with a formal specification of its elements, which supports the composition of self-adaptation mechanisms. The FORMS's static structure diagram specifies the types of elements required to implement adaptation mechanisms, and the relationships among these elements. Thus, self-adaptive implementations instantiated from FORMS are based on the MAPE-K loop, and rely on computational reflection approaches to affect managed elements. The MAPE components are realized as computations derived from meta-level computations, whereas the K component is realized in the form of models instantiated from meta-level models. Meta-level models are key enablers of adaptation mechanisms, which are supported by meta-level computations [34]. MAPE-K loop implementations, including hierarchical and decentralized compositions of MAPE-K loop components can be instantiated directly from FORMS. Therefore, FORMS addresses separation of concerns and visibility of the feedback loop's components in the same way as addressed by the autonomic manager. The FORMS model seems to be a suitable approach to implement self-adaptation mechanisms by exploiting model-driven engineering technologies. Nevertheless, implementations where the relevance of adaptation mechanisms and monitoring strategies must be controlled at runtime to address changes in adaptation objectives are not currently supported by FORMS.

The main contribution of our reference model refers to the separation of concerns required to deal with the three levels of dynamics in self-adaptation. In DYNAMICO, separation of concerns goes beyond the decoupling of adaptation mechanisms from managed systems. We introduce three different types of MAPE-K loops that must interact among them to address changes in self-adaptive approaches at three different levels: control objectives, adaptation, and monitoring. Our reference model characterizes the elements required to control adaptation mechanisms under highly changing execution conditions. These elements are the components of the three types of feedback loops, the control/data flow among their components, and the control/data flow among the three levels. DYNAMICO relies on the MAPE-K loop to characterize the components that define our control objectives, adaptation and monitoring feedback loops. However, DYNAMICO is independent of the particular strategies and technologies used for implementing self-adaptation. To characterize the elements of our reference model and the interactions among them, we analyzed 34 of the most representative research approaches to self-adaptation [21]. The surveyed approaches range from control theory-based approaches to pure software-based approaches. In control-based approaches, the managed system's structure is generally a non-modifiable structure, and control actions are continuous signals that affect behavioral properties of the managed system. In software-based approaches, the managed system's structure is commonly a modifiable structure, and control actions are discrete operations, supported by software models and reflection, that affect the system's software architecture. We proposed DYNAMICO to guide the design and implementation of dynamic control capabilities along the whole self-adaptive systems spectrum.

A software engineer can use DYNAMICO not only to instantiate, independently, each of the three feedback loops, but also to instantiate the interactions among these three feedback loops. Consequently, a DYNAMICO-based self-adaptive system can support changes in adaptation goals at runtime, as well as the use of these changes to adapt adaptation mechanisms and monitoring strategies accordingly. Moreover, this adaptive instrumentation can keep track of changes in monitoring strategies that could indicate the necessity of revising adaptation goals. Revisiting the application example used throughout this chapter, an adaptive solution purely based on any the feedback-loop, the MAPE-K loop, or the FORMS model could not adapt automatically the monitoring strategy after re-negotiating the performance SLA, according to the new context monitoring requirements stated with the throughput SLO (cf. Sect. 2.1).

According to Bass *et al.*, the process of designing a concrete software architecture for a system should start either from a reference model or an architectural style, or from both [14]. In either case, the process continues with successive refinement steps, where each step augments the previous one with additional information from further analysis of requirements in the problem domain, as well as global design decisions. In light of this, the application of DYNAMICO must be complemented with specific design patterns, architectural styles, design profiles, frameworks, and even other more specific or domain-dependent reference models for designing self-adaptive software systems. In particular, architectural patterns for interacting control loops such as the ones described in Sect. 4 of the first chapter in this book—the roadmap—constitute a suitable approach applicable to the design and implementation of feedback loop interactions defined in our reference model. Similarly, approaches such as the MAPE-K loop extensions proposed by Vromant *et al.* may be applied together with DYNAMICO, and selected architectural patterns to support intra- and inter-loop coordination during the different phases of self-adaptation [35]. Furthermore, due to its general nature, DYNAMICO supports the engineering of self-adaptive systems independently of concrete architectural considerations, such as the level of centralization or decentralization required by the control mechanism, as exemplified in [34]. UML profiles, such as the one proposed by Hebig *et al.* [36], provide valuable support for the design of UML-based concrete architectures based on our reference model.

6 Conclusions and Future Work

In this chapter we have presented DYNAMICO, a reference model for engineering highly dynamic adaptive software systems. This kind of system must deal with highly dynamic contexts of execution, and effectively respond to, by evaluating their own behaviour at runtime and reconfiguring itself whenever it no longer satisfies its requirements.

A highly dynamic context is characterized by (a) expected and unexpected changes in context conditions such as user location (in mobile software clients), network access point, service throughput and load (in the server side), time and calendar dates, and even user interests associated to specific locations and special dates; (b) dynamic changes in adaptation goals and user requirements, such as re-negotiation of QoS levels for specific services; and (c) other sensible changes that affect the satisfaction of system requirements, such as unauthorized intrusions or faults. In addition, all of these changes are assumed as natural requirements to be satisfied by the self-adaptive system at runtime.

DYNAMICO helps cope with this kind of dynamic requirements by defining three types of feedback loops. Each of these feedback loops manages each of the three levels of context dynamics that we characterized for self-adaptation: (i) the control objectives feedback loop, for managing changes in adaptation goals and user requirements; (ii) the target system adaptation feedback loop, to deal with changes addressable directly at the target system level; and (iii) the dynamic monitoring feedback loop, to manage changes that require the deployment of different or additional monitoring infrastructures to those already configured for execution, thus maintaining its relevance with respect to the changing adaptation goals. As a reference model, DYNAMICO reconciles the many visions and contributions of different approaches for the development of self-adaptive software systems, whether they hide or exhibit the elements of feedback control loops. Nonetheless, our reference model emphasizes the visibility of these control elements and constitutes a guide to design self-adaptive systems in which the system goals, the target system itself, or the monitoring infrastructure must be adapted—assuming this is a crucial requirement for the system to be developed. Depending on these requirements, the model can be applied as a whole, with its three feedback loops, or partially, involving only a subset of them.

We showed the applicability of DYNAMICO using a SOA governance application example based on an industrial case study. In this example, self-adaptation mechanisms are used to guarantee SLAs in a cloud-based infrastructure whose conditions of operation (i.e., efficiency and throughput) are re-negotiated at runtime, potentially compromising the effectiveness of the already deployed monitoring infrastructure. To reestablish the relevance of the monitoring infrastructure, we combined two of the three feedback loops managed in DYNAMICO: (i) the control objectives feedback loop for managing changes in the adaptation goals (i.e., SLAs); and (ii) the dynamic monitoring feedback loop to deploy the required additional monitoring elements.

For future research there are several opportunities for extending and validating DYNAMICO: (i) the use of DYNAMICO in additional validation cases, as part of the IBM CAS project "Managing Dynamic Context to Optimize Smart Interactions and Smart Services",[2] addressing different issues such as the dynamic discovery and adaptation of smart services to enable user-driven web integration, and supporting distributed feedback loops for decentralized adaptation control, as those discussed in Sect. 5.2; (ii) the concrete definition of control objectives as contracts, to support the synchronized cooperation between context management systems and self-adaptation mechanisms; (iii) the development of generalized governance infrastructures to manage feedback loop interactions; and (iv) the definition of a formal framework to evaluate and compare adaptation mechanisms based on the three levels of self-adaptation dynamics that we characterized in Sect. 3.3. For this, our characterization model and adaptation properties can be used as a useful starting point [33].

[2] https://www-927.ibm.com/ibm/cas/cassis/viewReport?REPORT=747

Acknowledgments. This work was funded in part by the National Sciences and Engineering Research Council (NSERC) of Canada under the Strategic Networks Grants Program (NETGP 397724-10) and Collaborative Research and Development program (CRDPJ 320529-04 and CRDPJ 356154-07), IBM Corporation, CA Inc., Icesi University (Cali, Colombia), and Ministry of Higher Education and Research of Nord-Pas de Calais Regional Council and FEDER under Contrat de Projets Etat Region (CPER) 2007-2013.

References

1. 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)
2. United States Air Force Chief Scientist (AF/ST): Technology Horizons a Vision for Air Force Science & Technology During 2010-2030. Technical report, U.S. Air Force (2010)
3. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
4. Truex, D.P., Baskerville, R., Klein, H.: Growing Systems in Emergent Organizations. Communications of the ACM 42(8), 117–123 (1999)
5. Tran, V.X., Tsuji, H.: A Survey and Analysis on Semantics in QoS for Web Services. In: International Conference on Advanced Information Networking and Applications, pp. 379–385. IEEE (2009)
6. Tamura, G., Casallas, R., Cleve, A., Duchien, L.: QoS Contract-Aware Reconfiguration of Component Architectures Using E-Graphs. In: Barbosa, L.S. (ed.) FACS 2010. LNCS, vol. 6921, pp. 34–52. Springer, Heidelberg (2010)
7. Oreizy, P., Medvidovic, N., Taylor, R.N.: Runtime Software Adaptation: Framework, Approaches, and Styles. In: 30th International Conference on Software Engineering (ICSE 2008), pp. 899–910 (2008)
8. Giese, H., Brun, Y., Serugendo, J.D.M., Gacek, C., Kienle, H., Müller, H., Pezzè, M., Shaw, M.: Engineering Self-Adaptive and Self-Managing Systems. LNCS 5527, 47–69. Springer (2009)
9. Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D.M.: Feedback Control of Computing Systems. John Wiley & Sons (2004)
10. Müller, H.A., Kienle, H.M., Stege, U.: Autonomic Computing Now You See It, Now You Don't—Design and Evolution of Autonomic Software Systems. In: De Lucia, A., Ferrucci, F. (eds.) ISSSE 2006-2008. LNCS, vol. 5413, pp. 32–54. Springer, Heidelberg (2009)
11. IBM Corporation: An Architectural Blueprint for Autonomic Computing. Technical report, IBM Corporation (2006)
12. Weyns, D., Malek, S., Andersson, J.: FORMS: a FOrmal Reference Model for Self-adaptation. In: 7th International Conference on Autonomic Computing, ICAC 2010, pp. 205–214. ACM, New York (2010)

13. Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. In: 2007 Workshop on the Future of Software Engineering (FOSE 2007), pp. 259–268. IEEE Computer Society (2007)
14. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley, Reading (2003)
15. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. Computer 36(1), 41–50 (2003)
16. Papazoglou, M.P., Heuvel, W.J.: Service Oriented Architectures: Approaches, Technologies and Research Issues. The Very Large Databases (VLDB) Journal 16, 389–415 (2007)
17. Villegas, N.M., Müller, H.A., Tamura, G.: Optimizing Run-Time SOA Governance through Context-Driven SLAs and Dynamic Monitoring. In: 2011 IEEE International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA 2011), pp. 1–10. IEEE (2011)
18. Villegas, N.M., Müller, H.A.: Context-driven Adaptive Monitoring for Supporting SOA Governance. In: 4th International Workshop on a Research Agenda for Maintenance and Evolution of Service-Oriented Systems (MESOA 2010). CMU/SEI-2011-SR-008, Pittsburgh: Carnegie Mellon University (2011)
19. Lee, J.Y., Lee, J.W., Cheun, D.W., Kim, S.D.: A Quality Model for Evaluating Software-as-a-Service in Cloud Computing. In: 7th ACIS International Conference on Software Engineering Research, Management and Applications (SERA 2009), pp. 261–266. IEEE Computer Society, Washington, DC (2009)
20. Müller, H., Pezzè, M., Shaw, M.: Visibility of Control in Adaptive Systems. In: 2nd International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008), pp. 23–26 (2008)
21. 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, pp. 80–89. ACM, New York (2011)
22. Shaw, M.: Beyond Objects: A Software Design Paradigm Based on Process Control. ACM Software Engineering Notes 20(1), 27–38 (1995)
23. Villegas, N.M., Müller, H.A.: Managing Dynamic Context to Optimize Smart Interactions and Services. In: Chignell, M., Cordy, J., Ng, J., Yesha, Y. (eds.) The Smart Internet. LNCS, vol. 6400, pp. 289–318. Springer, Heidelberg (2010)
24. Bianco, P., Lewis, G., Merson, P.: Service Level Agreements in Service-Oriented Architecture Environments. Technical Report CMU/SEI-2008-TN-021, CMU/SEI (2008)
25. Elkhodary, A., Esfahani, N., Malek, S.: FUSION: a Framework for Engineering Self-Tuning Self-Adaptive Software Systems. In: 18th ACM International Symposium on Foundations of Software Engineering, FSE 2010, pp.7–16. ACM (2010)
26. Dumont, G., Huzmezan, M.: Concepts, Methods and Techniques in Adaptive Control. In: 2002 American Control Conference, vol. 2, pp. 1137–1150. IEEE (2002)
27. Narendra, K.S., Balakrishnan, J.: Adaptive Control Using Multiple Models. IEEE Transactions on Automatic Control 42, 171–187 (1997)
28. Garlan, D., Cheng, S.W., Schmerl, B.: Increasing System Dependability through Architecture-based Self-Repair. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems. LNCS, vol. 2677, pp. 61–89. Springer, Heidelberg (2003)
29. Parra, C., Blanc, X., Duchien, L.: Context Awareness for Dynamic Service-Oriented Product Lines. In: 13th Intentaional Software Product Line Conference (SPLC 2009), pp. 131–140 (2009)

30. Solomon, B., Ionescu, D., Litoiu, M., Mihaescu, M.: A Real-time Adaptive Control of Autonomic Computing Environments. In: 17th Annual International Conference hosted by the Centre for Advanced Studies Research, IBM Canada Software Laboratory (CASCON 2007), pp. 124–136 (2007)
31. Capraresc u, B.A., Petcu, D.: A Self-Organizing Feedback Loop for Autonomic Computing. Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, Computation World, 126–131 (2009)
32. Gat, E.: Three-layer Architectures. MIT Press, Cambridge (1998)
33. Frîincu, M.E., Villegas, N.M., Petcu, D., Müller, H.A., Rouvoy, R.: Self-Healing Distributed Scheduling Platform. In: 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2011, pp. 225–234. IEEE Computer Society, Washington, DC (2011)
34. Weyns, D., Malek, S., Andersson, J.: On Decentralized Self-Adaptation: Lessons from the Trenches and Challenges for the Future. In: 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010), pp. 84–93. ACM, New York (2010)
35. Vromant, P., Weyns, D., Malek, S., Andersson, J.: On Interacting Control Loops in Self-Adaptive Systems. In: 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011), pp. 202–207. ACM, New York (2011)
36. Hebig, R., Giese, H., Becker, B.: Making control loops explicit when architecting self-adaptive systems. In: 2nd International Workshop on Self-Organizing Architectures, SOAR 2010, pp. 21–28. ACM, New York (2010)

Fault-Adaptivity in Hard Real-Time Component-Based Software Systems*

Abhishek Dubey, Gabor Karsai, and Nagabhushan Mahadevan

Institute for Software-Integrated Systems,
Vanderbilt University,
Nashville, TN 37203, USA

Abstract. Complexity in embedded software systems has reached the point where we need run-time mechanisms that provide fault management services. Testing and verification may not cover all possible scenarios that a system encounters, hence a simpler, yet formally specified run-time monitoring, diagnosis, and fault mitigation architecture is needed to increase the software system's dependability. The approach described in this paper borrows concepts and principles from the field of 'Systems Health Management' for complex aerospace systems and implements a novel two level health management architecture that can be applied in the context of a model-based software development process.

At the first level, the Component-level Health Manager (CLHM) provides localized and limited service for managing the health of individual software components. A higher-level System-level Health Manager (SLHM) manages the health of the overall system. SLHM includes a diagnosis engine that uses a Timed Failure Propagation (TFPG) model automatically synthesized from the system specification built in the model-based design environment that accompanies the runtime system. SLHM also includes a reactive timed state machine used for mitigation, whose code is also generated from the model-based specification. This paper uses simple examples to illustrate the use of the approach.

1 Introduction and Motivation

Software has become the key enabler for a number of core capabilities and services in modern systems [28]. For example, a modern car contains around 20 million lines of code, while just the flight control software of modern aircraft like F-22 and F-35 contains $1.7 - 5.7$ million lines of code [9]. Given the scale of the software systems, it is not hard to appreciate the challenge of ensuring correct behavior, especially in avionics where software malfunctions have caused a number of incidents in the

* This paper is based upon work supported by NASA under award NNX08AY49A. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Aeronautics and Space Administration. The authors would like to thank Dr Paul Miner, Eric Cooper, and Suzette Person of NASA LaRC for their help and guidance on the project.

R. de Lemos et al. (Eds.): Self-Adaptive Systems, LNCS 7475, pp. 294–323, 2013.

past, including but not limited to those referred to in these reports: [5,6,18,29]. [36] provides an excellent discussion on the complexity in avionics software.

The state of the art for critical software development includes process standards such as DO-178B [12] and the emerging standards such as DO-178C [21]. However, it is known that software can contain latent defects or bugs that can escape the existing rigorous testing and verification techniques and manifest only under exceptional circumstances. These circumstances may include faults in the hardware system, including both the computing and non-computing hardware. Often, systems are not prepared for such faults.

State of the art for safety critical systems is to employ software fault tolerance techniques that rely on redundancy and voting [8,25,40]. However, it is clear that existing techniques do not provide adequate coverage for problems such as common-mode faults and latent design bugs triggered by other faults. Additional techniques are required to make the systems self-managing, i.e. they have to provide resilience to faults by adaptively mitigating the functional effects of those faults.

Self-adaptive systems must be able to adapt to faults in software as well as the hardware (physical equipment) elements of a system, even if they appear simultaneously. Conventional Systems Health Management is associated with the physical elements of the system, and includes anomaly detection, fault source identification (diagnosis), fault effect mitigation (at runtime/ online during operation), maintenance (offline), and fault prognostics (online or offline) [22,30]. Software Health Management (SHM), borrows concepts and techniques from Systems Health Management and is a systematic extension of classical software fault tolerance techniques. Srivastava and Schumann provide a good motivation for Software Health Management in [38]. SHM is performed at run-time, and just like Systems Health Management it includes detection, isolation, and mitigation to remove fault effects. SHM can be considered as a dynamic fault removal technique [4]. While Systems Health Management also includes prognostics, Software Health Management could possibly be extended in that direction as well, but we have not investigated it yet.

We have developed an approach and model-based support tools for implementing software health management functions for component-based systems. The foundation of the architecture is a real-time component framework that defines a component model for ARINC-653 systems[1] [14]. This framework brings the concept of temporal isolation, spatial isolation, strict deadlines from ARINC-653 and merges it with the well-defined interaction patterns described in CORBA Component Model [42]. The health management in the framework is performed at two levels. The Component-level Health Manager (CLHM) provides localized and limited service for managing the health of individual software components. A higher-level System Health Manager (SLHM) manages the health of the overall system.

[1] ARINC-653 (Avionics Application Standard Software Interface) is a specification for space and time partitioning in Safety-critical avionics Real-time operating systems. It allows to host multiple applications of different software levels on the same hardware in the context of an Integrated Modular Avionics architecture.[1,32].

SLHM includes a diagnosis engine that uses a Timed Failure Propagation (TFPG) model automatically synthesized from the component assembly; the engine reasons about fault effect cascades in the system, and isolates the fault source components. This is possible because the data / behavioral dependencies and hence the fault propagation across the assembly of software components can be deduced from the well-defined and restricted set of interaction patterns supported by the framework. Once the fault source is isolated, the necessary system level mitigation action is taken. Similar approaches can be found in [23,41]. The key difference between those and our work is that we apply an online diagnosis engine coupled with a two-level mitigation scheme. Furthermore, this approach is applied to hard real-time systems where all processes run within finite time bounds and are continuously monitored for deadline violations. This includes, the health management processes.

Our approach is supported by a model-based design environment where developers can create models of the system and its components, as well as specify how fault mitigation will take place. A suite of software generators produce glue code that allows developer-supplied functional code or 'business logic' to form a collection of applications that run on an ARINC-653 platform. The novel contributions of our approach are:

- Model-based development of component-based systems for ARINC-653 platform.
- Automatic synthesis of monitoring code that is executed with the component operations.
- Automatic synthesis of diagnosis information from the system design models.
- Automatic synthesis of the mitigation code based on system specification.
- Generation and configuration of the distributed architecture required to operate the components in parallel with the component and system level health managers.

This paper is an extended version of the work presented in [15,27]. It uses simple examples to describe the approach. A larger case study of applying the SHM principles to an Inertial Measurement Unit is available as a technical report [16]. In this paper, the focus is on the mitigation aspects: the support provided in the framework and modeling language. The outline of this paper is as follows: Sections 2 discusses the related research. Overview of the component model and design tools is given in Section 3. Section 4 presents component health manager and system-level health manager. Finally we conclude with discussions and future work.

2 Related Research and Background

The work described here fits in the general area of self-adaptive software systems, for which a research roadmap has been presented in [10]. Our approach focuses on latent faults in software systems, follows a component-based architecture, with a model-based development process, and implements all steps in the Collect/ Analyze/Decide/Act loop. In this context of health management, this would imply *Collect* details about anomalies observed), identify/ diagnose the fault candidate, and *Decide* on the possible mitigation command and finally *Act* to implement the mitigation commands.

Rohr et al. advocate the use of architectural models for self-management [35]. They suggest the use of a runtime model to reflect the system state and provide reconfiguration functionality. From a development model they generate a causal graph over various possible states of its architectural entities. At the core of their approach, they use specifications based on UML to define constraints, monitoring and reconfiguration operations at development time.

Garlan et al. [17] and Dashofy et al. [11] have proposed an approach which bases system adaptation on architectural models representing the system as a composition of several components, their interconnections, and properties of interest. Their work follows the theme of Rohr et al., where architectural models are used at runtime to track system state and make reconfiguration decisions using rule-based strategies.

While these works have tended to the structural part of the self-managing computing components, some have emphasized the need for behavioral modeling of the components. For example, Zhang et al. described an approach to specify the behavior of adaptable programs in [46]. Their approach is based on separating the adaptation behavior specification from the non-adaptive behavior specification in autonomic computing software. They model the source and target models for the program using state charts and then specify an adaptation model, i.e., the model for the adaptation set connecting the source model to the target model using a variant of Linear Temporal Logic [45].

Williams' research [34] concentrates on model-based autonomy. The paper suggests that emphasis should be on developing techniques to enable the software to recognize that it has failed and to recover from the failure. Their technique lies in the use of a Reactive Model-based Programming Language (RMPL)[43] for specifying both correct and faulty behavior of the software components. They also use high-level control programs [44] for guiding the system to the desirable behaviors.

Lately, the focus has started to shift to formalize the software engineering concepts for self-management. In [24], Lightstone suggested that systems should be made "just sufficiently" self-managing and should not have any unnecessary complicated function. Shaw proposes a practical process control approach for autonomic systems in [37]. The author maintains that several dependability models commonly used in autonomic computing are impractical as they require precise specifications that are hard to obtain. It is suggested that practical systems should use development models that include the variability and unpredictability of the environment. Additionally, the development methods should not pursue absolute correctness (regarding adaption) but should rather focus on the fitness for the intended task, or sufficient correctness. Several authors have also considered the application of traditional requirements engineering to the development of autonomic computing systems [7,39].

The work described here is closely related to the larger field of software fault tolerance: principles, methods, techniques, and tools that ensure that a system can survive software defects that manifest themselves at run-time [26,33]. Arguably, our approach comes closest to dynamic software fault removal, performed at run-time. The overall architecture presented below shows a specific implementation of the functions needed to perform this task.

3 Overview of ARINC-653 Component Model

Systems health management and fault tolerance approaches are based on the notion of interacting components. Hence, it is natural to apply this concept to SHM, where the software is built from components that can be individually developed, monitored and managed at run-time. In our work, the first step was to develop and implement such a component model. The ARINC-653 component model (ACM) is built upon the services of ARINC-653; an avionics standard for safety critical operating systems [1]. ARINC-653 systems group *processes*[2] into spatially and temporally separated *partitions*, with one or more partitions assigned to each *module* (i.e. a processor), and one or more modules forming a *system.*

Spatial partitioning ensures exclusive use of a memory region by an ARINC-653 partition. It also guarantees that a faulty process in a partition cannot ruin the data structures of other processes in other partitions, isolating low-criticality vehicle management components from safety-critical flight control software components. Temporal partitioning ensures exclusive use of the processing resources by a partition. A fixed periodic schedule is used by the RTOS to share the resources between partitions. This deterministic scheduling ensures that each partition is allowed exclusive access to the processor or other hardware resources within its predetermined execution interval. It also guarantees that when the predetermined execution interval of a partition is over, the partition's execution will be interrupted, the partition will be placed into a dormant state and the next partition in the schedule order will be granted exclusive access to the computing resource, i.e. the processor.

The ARINC-653 Component Model (ACM) allows the developers to group a number of ARINC-653 processes into a reusable component. A component is a group of processes that share state but they do not interact directly. However, components do interact with each other via well-defined interaction patterns (chosen from a fixed set), facilitated by ports. In ACM, a component can have four kinds of external ports for interactions: **publishers, consumers, facets** (provided interfaces[3]) and **receptacles** (required interfaces), see Figure 1. Each port has an interface type (a named collection of methods) or an event type (a data structure). The component can interact with other components through **synchronous** call/return interfaces (associated with facets or receptacles), and/or via **asynchronous** publish/subscribe event connections (assigned to publisher and consumer ports). Additionally, a component can host internal methods that are periodically triggered. Most of these interactions borrow concepts from other software component frameworks, notably from the CORBA

[2] An ARINC-653 process is a unit of concurrency that is analogous to thread in a desktop operating system such as Linux.

[3] An interface is a collection of related methods.

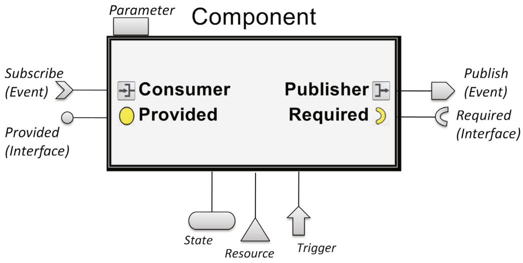


Fig. 1. The Component Model

Component Model (CCM) [42]. The component model also provides guidance on the allocation of activities to a component.

Real-Time Properties. ACM components differ from classical CCM component in a number of ways. The underlying operating system layer on which ACM is built is geared towards hard real-time systems. Therefore, in ACM all processes have fixed properties that are specified and fixed at the system configuration time - these properties include, period, deadline, stack size and priority. Furthermore, a process can have two kinds of deadlines, HARD deadline and SOFT deadline. A HARD deadline violation is an error that is handled at the system level by the health management framework, discussed later in the paper. A soft deadline violation results in warnings. Due to these restrictions, it is not possible to dynamically assign component operations or ports to an ARINC-653 system at runtime. Therefore, all ports are statically bound to an ARINC-653 process and no dynamic memory allocation is allowed. Furthermore, the access to component state is synchronized by a component-wide lock. Priority inversion issues do not arise because all processes of a component are executed at the same priority. Please see [14] for detailed description.

The framework implementing ARINC-653 component model consists of two parts (a) a Linux-based runtime environment, and (b) a modeling environment and associated design tools. Together these tools allow systems to be developed in two distinct phases. The first phase is completed by the component developer. A component is a reusable artifact that provides one or more functionalities. It can be developed and hosted in a repository for reuse. Often, the component developer can organize various components into subsystems. The second phase is completed by the system integrator. The system integration includes the modeling and configuring of the system architecture, deploying the components on computing hosts, etc. This phase is assisted by a suite of model-driven tools.

3.1 Component Development

The model-based design tools[4] allow the developer to design the components. The first step in designing a component involves defining its interfaces, i.e. the

[4] These tools are available for download from `https://wiki.isis.vanderbilt.edu/mbshm/index.php/Main_Page`

ports associated with the component. Each port, as described earlier, should belong to one of the four categories: publisher, consumer, provided, required. The publisher and consumer ports need to be associated with an event (data) type, while provided and required ports need to be associated with an interface type. Furthermore, each port needs to be configured with properties related to its real-time execution: periodicity, deadline, worst case execution time, etc.

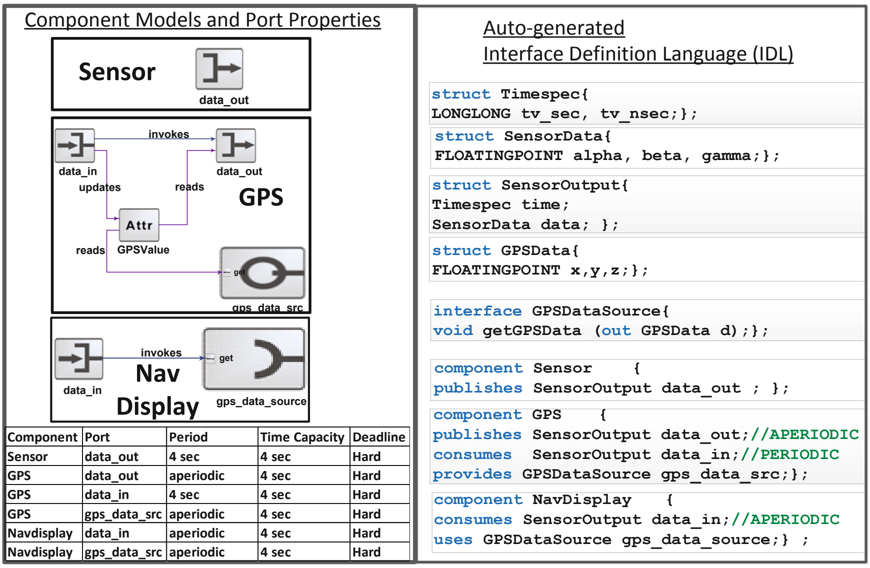
The development environment provides tool-support to bring the bare-bones component model to life. As a first step, a C++ class is generated corresponding for each component, with methods corresponding to each port. The developer is provided with specific regions in the generated code to insert the necessary code and logic to customize the behavior of each port (as per the associated task). The generated code acts as the 'glue' between the underlying ACM framework and the user-specified code to support the operation/execution of each of the component ports as dedicated ARINC-653 processes.

The component model can be further enriched by specifying the conditions that must be satisfied for each execution of the port (and its associated ARINC-653 process). These conditions are divided into three categories: pre-conditions, invariants, and post-conditions. The design tools generate monitoring code that is used to ensure and validate the correctness of these conditions during runtime. Any violation of these conditions is considered an **anomaly**. More discussion on this topic will be provided later in Section 3.2. Each component developer can also specify a local mitigation activity: a component level health manager that takes local corrective actions when an anomaly is detected. Once fully specified, the component model captures the component's interaction ports, conditions associated with the ports, the real-time properties and resource requirements of the ports and the component, the data and control flow within the component, and (optionally) the local component level health management strategy.

Example. Figure 2 shows three components developed in ACM modeling environment. It also shows portion of the Interface Definition Language (IDL) file generated by the associated tools. The first component is the sensor component that publishes a data type “SensorOutput” periodically every 4 sec. The second component is the GPS component that receives the input from a Sensor, then filters it, and updates its internal data structure. It publishes the updated information through a port aperiodically. The GPS has the ability to be queried remotely via a method call for the current GPS value. The last component is a Navigation Display component, which receives an updated SensorOutput and also queries a remote GPS interface. It should be noted that the components described were developed in isolation, i.e. they are developed as part of a distributed system.

3.2 Component Execution and Failure Scenarios

Any component, once deployed in the system can be in one of the following three states: **active**, **inactive** and **semi-active**. When a component is in inactive state, none of the ports in the Component perform their task. The active state



Component	Port	Period	Time Capacity	Deadline
Sensor	data_out	4 sec	4 sec	Hard
GPS	data_out	aperiodic	4 sec	Hard
GPS	data_in	4 sec	4 sec	Hard
GPS	gps_data_src	aperiodic	4 sec	Hard
Navdisplay	data_in	aperiodic	4 sec	Hard
Navdisplay	gps_data_src	aperiodic	4 sec	Hard

```
struct Timespec{
LONGLONG tv_sec, tv_nsec;};
struct SensorData{
FLOATINGPOINT alpha, beta, gamma;};
struct SensorOutput{
Timespec time;
SensorData data; };
struct GPSData{
FLOATINGPOINT x,y,z;};

interface GPSDataSource{
void getGPSData (out GPSData d);};

component Sensor    {
publishes SensorOutput data_out ; };
component GPS    {
publishes SensorOutput data_out;//APERIODIC
consumes  SensorOutput data_in;//PERIODIC
provides GPSDataSource gps_data_src;};
component NavDisplay    {
consumes SensorOutput data_in;//APERIODIC
uses GPSDataSource gps_data_source;} ;
```

Fig. 2. Components developed using ACM Design Tools. This figure contains the internal ports of the components, including the internal data flow and control flow. Also shown are the snapshots of generated Interface Definition Language (IDL) files and the associated real-time properties for each port.

of a component is the exact opposite of the inactive, and all the component ports perform their task. In a semi-active state, only the Consumer and Receptacle ports of a component are operational, the Publisher and Provided ports are disabled. During nominal operation, a component is either in the active state, or semi-active state. The semi-active state is typically assigned to passive replicas, if any, in the system by the system integrator. Typically, a component is made inactive only if it is diagnosed as faulty at runtime.

While the component is executing i.e. it is in active or semi-active state, component ports can introduce faults in the system. We consider two root failure sources for each component port (a) a concurrency fault: caused by the timeout in the act of obtaining the lock associated with the component, (b) a latent defect in the code written by the developer for handling the activity of the port.

Both of the above fault scenarios can lead to several secondary anomalies in either the same component or in a connected component. In our framework, the design tools allow the system designer to specify monitors which can be configured to detect deviations from expected behavior, violations of specifications and conditions of an interaction port or component. Based on these monitors, following discrepancies can be currently identified:

- *Lock timeout*: The framework implicitly generates monitors to check for resource starvation. Each component has a lock (to avoid interference among callers), and if a caller does not get through the lock within a specified time, an anomaly is declared. The value for timeout is either set to a default value equal to the deadline of the process associated with component port or can be specified by the system designer.

- *Data validity violation* (only applicable to consumers): Any event data token consumed by a consumer port has an associated expiration age. This is also known as the validity period in ARINC-653 sampling ports. We have extended this to be applicable to all types of component consumer ports, both periodic and aperiodic.
- *Pre-condition violation*: Developers can specify conditions that should be checked before executing. These conditions can be expressed over the current value or the historical change in the value, or rate of change of values of variables (with respect to previously known value for same parameter) such as
  1. the event data of asynchronous calls,
  2. function parameters of synchronous calls, and
  3. (monitored) state variables of the component.
- *User-code failure*: Any error or exception raised in the user code can be abstracted by the software developer as an error condition which can then be reported to the framework. Any unreported error is recognized as a potential unobservable discrepancy.
- *Post-condition violation*: Similar to pre-condition violations, but these conditions are checked after the execution of the function associated with the component port.
- *Deadline violation*: Any process execution must finish within the specified deadline.

These monitors can be specified via (1) attributes of model elements (e.g. Deadline, Data_Validity, Lock time out), and (2) via a simple expression language. The expressions can be formed over the (current) values of variables (parameters of the call, or state variables of the component), their *change* (delta) since the last invocation, their *rate* of change (change divided by a time interval). Table 1 provides the summary of anomalies that can be observed on a component port and the component as a whole. Code generators included in the design tools generate the appropriate code for the monitors. While most of the monitors described above are evaluated in the same thread executing the component port, the monitors associated with resource usage (i.e. CPU time) are run in parallel by framework. Figure 3 shows the flowchart of the code generated to handle incoming messages on a consumer port. The failed monitored discrepancy is always reported to the local component health manager. Deadline violation is always monitored in parallel by the runtime framework.

Note 1. It is necessary to point out that the pre-conditions and post-conditions, if specified, should be verified against the formal system specification. We argue that it is easier to verify these conditions at run-time compared to formally verifying the full system. However, the full system should undergo rigorous testing[5].

[5] While formal verification covers all the possible behavior and environment interleaving traces, testing only covers the subset of all possible traces.

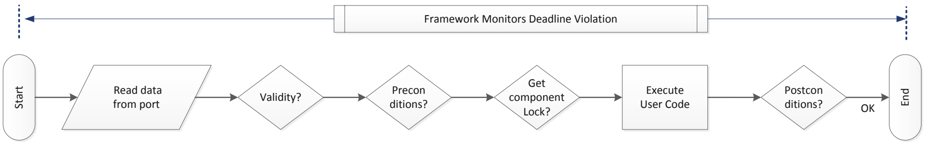


Fig. 3. Flow chart describing the interleaving of monitor and business logic provided by the user for a consumer port. The generated sequence is similar for other ports.

Table 1. Monitoring Specification. Comments are shown in italics.

<**Pre-condition**>::=<Condition>
<**Post-condition**>::=<Condition>
<**Deadline**>::=<double value> _/* from the start of the process associated with the port to the end of that method */_
<**Data_Validity**>::=<double value> _/* Max age from time of publication of data to the time when data is consumed*/_
<**Lock timeout**>::=<double value> _/* from start of obtaining lock*/_

<Condition>::=<Primitive Clause><op><Primitive Clause>|<Condition><logical op><Condition>| !<Condition> | True| False
<Primitive Clause>::=<double value>| Delta(Var)| Rate(Var)|Var _/* A Var can be either the component State Variable, or the data received by the publisher, or the argument of the method defined in the facet or the receptacle*/_
<op>::= < | > | <= | >= | == | !=
<logical op>::=&& | ||

During runtime, the formally verified conditions provide a blueprint for ensuring that the system/components are working without any discrepancy.

Example. In the GPS assembly shown in Figure 4, the ACM ports are configured with monitors of different kinds. Publisher and Consumer ports in Sensor, GPS, and NavDisplay are configured with monitors to track any violation of CPU resource usage (detected as Deadline Violation), the Publisher port in the Sensor component is configured to detect any violations/ problems with the user code (detected as User Code Violation), the Consumer ports in GPS and NavDisplay are configured to monitor problems with the age of the received data (detected as Data-Validity Violation), and Consumer and Receptacle port in NavDisplay are configured to detect Post-condition Violations.

3.3 System Integration

The modeling tools allow the system integrator to construct a system model by using the library of component models created by component developers. The modeling tool allows the system integrator to define the functionalities expected in the system and identify the appropriate components to provide these functionalities. The integrator creates the assembly model by instantiating and connecting the components, thereby capturing the interactions across the component assembly. At this time, the design constraints in the tools ensure that all

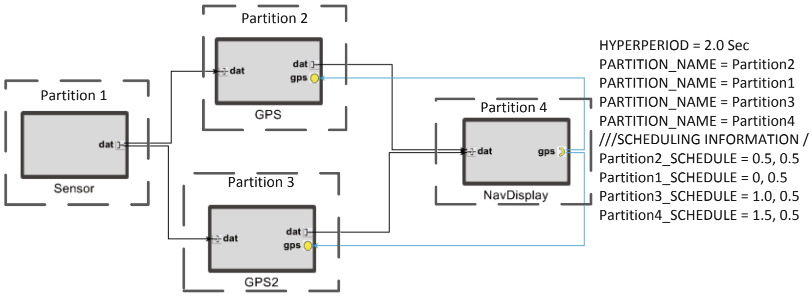

Partition 2
dat
dat
gps
GPS
Partition 1
dat
Sensor
Partition 3
dat
dat
gps
GPS2
Partition 4
dat
gps
NavDisplay
HYPERPERIOD = 2.0 Sec
PARTITION_NAME = Partition2
PARTITION_NAME = Partition1
PARTITION_NAME = Partition3
PARTITION_NAME = Partition4
///SCHEDULING INFORMATION /
Partition2_SCHEDULE = 0.5, 0.5
Partition1_SCHEDULE = 0, 0.5
Partition3_SCHEDULE = 1.0, 0.5
Partition4_SCHEDULE = 1.5, 0.5

Fig. 4. Example: GPS Software Assembly. Unit of time is seconds.

ports are properly connected, e.g. the type of publisher matches the subscriber. The modeling tool also allows the system integrator to organize connected components into subsystems, which could then be reused to build more complex assemblies.

Once the assembly is specified logically the integrator can model the details of the platform and capture the deployment information. The modeling tool allows the specification of the platform in terms of the modules (i.e. processors) and the ARINC-653 partitions within each module. The integrator can specify the deployment of each component into an appropriate partition such that the temporal partitioning concerns are satisfied. At this time the integrators can use the integrated system model (assembly, platform, deployment models) to perform an end-to-end timing study on the system to check the logical correctness of design. Design tools are also used to fully generate the integration code and configuration files. These tools also generate the required build system along with necessary files to use the Eclipse IDE for final compilation and editing.

Example. Figure 4 shows the integration model for the three GPS components showed in Figure 2. This model shows the connection between the components and their deployment on four different partitions. Partition 1 contains the Sensor Component. Partition 2 contains the GPS and Partition 4 contains the Navigation Display component. The sensor component publishes an event every 4 sec. The GPS component consumes the event published by sensor at a periodic rate of 4 sec. Afterwards it publishes an event, which is sporadically consumed by the Navigation Display (abbreviated as display). Thereafter, the display component updates its location by using getGPSData facet of the GPS Component. The publisher-consumer interaction between sensor and GPS components is implemented via a sampling port (Sampling ports are basic inter-partition communication mechanism in ARINC-653 platforms). A Channel connects the source sampling port from partition 1 to destination sampling port in partition 2. In this example, a redundant GPS is also connected in the assembly. The redundant component in this case shares the port structure with the other GPS. However, their internal behaviors are different. In this particular example, GPS 2 is set to the semi-active State i.e. it can consume but not publish.

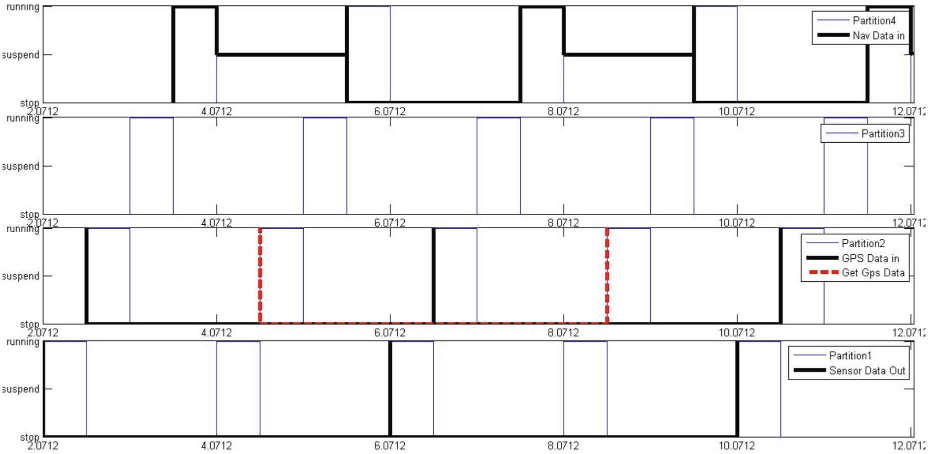


Fig. 5. Timing diagram for execution in the example

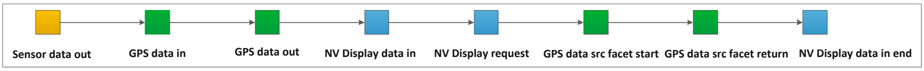


Fig. 6. The Chain of Events associated with data production and consumption across components in a hyper period

Figure 4 also describes the periodic schedule followed by the partitions, overseen by a controller process called Module Manager [14]. This schedule is repeated every 2 s (hyper period). In each cycle, Partition 1 runs with a phase of 0 sec for 500 ms. (duration). Partition 2's phase is 500 ms. It runs for 500 ms. Then Partition 3 and Partition 4 run for next 1 second. This schedule ensures that the two partitions are temporally isolated. Figure 6 shows the timing diagram. Notice that the partitions are temporally isolated from each other. Ports are suspended when their partition is context switched. The GPS _data_in and Sensor_Data_out's execution time is very low. That is why they appear as impulses on the graph. The NavDisplay data in consumer takes longer to run because it invokes the Receptacle port to send a synchronous request to the GPS facet port, which cannot be fulfilled until GPS's partition becomes active.

Note 2. The temporal isolation and partition time allocation in this architecture is strict, i.e. activities in one partition do not affect activities in another partition unless there is an explicit data dependency. Even with data dependencies the time allocated to partitions remains as specified during design. Another point to note is that due to the use of model-driven tools and auto generation of the integration code, the system integrator can quickly change the deployment scenario by allocation each component to a different partition and regenerating the code. However, such a change requires the recompilation of affected partitions and can have an effect on the timing of the system.

4 Health Managers

In component-based systems, anomalies in a component can be either local or secondary effect of an anomaly in an upstream component. Identifying this pattern is important in order to isolate the root failure source. While the component level mitigation code (provided by a component developer) can quickly react to the local anomaly and possibly arrest any problems that could arise because of the anomaly, this mitigation action may not remove the primary source of failure. A system wide response/ mitigation engine would be ill-suited to react to every local anomaly but would be better positioned to identify and mitigate the real-fault source, especially when the failure effects cascade across component boundaries. Realizing the benefits and limitations of each strategy, we implemented a two level health management strategy in our framework with a *component level* that is local to a component, and the *system level* that covers the entire assembly of components. While the component level health manager is specified by the component developers, the system level health manager is provided by the system integrator. Both health managers are specified as hierarchical timed state machines using the modeling tools. Please refer to [15] for a formal description of these state machines.

Code generators are responsible for mapping the specified management logic to the runtime system, which ensures that the specified state machine logic is executed using a variation of the Harel state chart semantics [19]. Discussion of these semantics is not included in this paper. It should be noted that these managers are reactive because they are triggered by either an event, e.g. occurrence of an anomaly, or passage of time (i.e. a timeout). When triggered, the machine reevaluates its current state and in the process executes actions specified on the transition or for the state (entry, exit, or during). The next two sections describe both the component and system level health managers.

4.1 Component Level Health Manager

Component-Level Health Manager(CLHM) provides localized and limited functionality for managing the health of the internals of a component. The health manager reacts with appropriate mitigation action to the anomalies detected within the component. As described in the previous section, CLHM is implemented as a timed state machine. It is triggered by anomalies detected by the monitors deployed inside the component, as shown in Table 1.

In addition to these monitors that detect and report anomalies, monitors to report *ENTRY* into and *EXIT* out of a port's process can also be specified using the modeling tool. These monitors aid in building observer models to track the execution sequence of component processes (ports) and report any deviations from the expected sequence. Observers are modeled as parallel state machines within the CLHM with one machine acting as an observer and another as the health manager. Each of the parallel state machines could be triggered by their relevant monitor events. While the observer tracks the state evolution, the health manager issues appropriate mitigation action for the anomalies detected. When

Table 2. CLHM Mitigation Actions

CLHM Action	**Semantics**
IGNORE()	Continue as if nothing has happened
ABORT()	Discontinue current operation, but operation can run again
USE_PAST _DATA()	Use most recent data (only for operations that expect fresh data)
STOP(p)	p is the process id. Default value is current process. This commands discontinues operation in process 'p'. Aperiodic processes (ports): operation can run again Periodic processes (ports): operation must be enabled by a future START HM action
START(p)	'p' is the same as defined in context of STOP (above). Re-enable a STOP-ped periodic operation
RESTART(p)	'p' is the same as defined in context of STOP (above). A Macro for STOP followed by a START for the current operation

an anomaly is detected in the observer, it triggers the health manager portion of the CLHM state machine to take the appropriate mitigation action.

The mitigation commands that can be expressed in the timed state machine model for CLHM are described in the Table 2. These commands can be issued as a transition action (executed when a state transition succeeds) or as entry, exit or during action of a state.

The CLHM associated with each component is hosted on a separate high priority ARINC-653 process. When a monitor reports a violation, the report is communicated to the relevant manager using the methods supported by the framework, e.g. an ARINC-653 buffer, see Figure 7. The buffer provides an intra-partition FIFO message queue for communication. This rerport triggers the execution of the CLHM state machine code which responds with an appropriate mitigation action. Depending on the nature of the mitigation action, the appropriate command is communicated either to the framework or to a relevant process which then executes it. Commands such as IGNORE, ABORT,

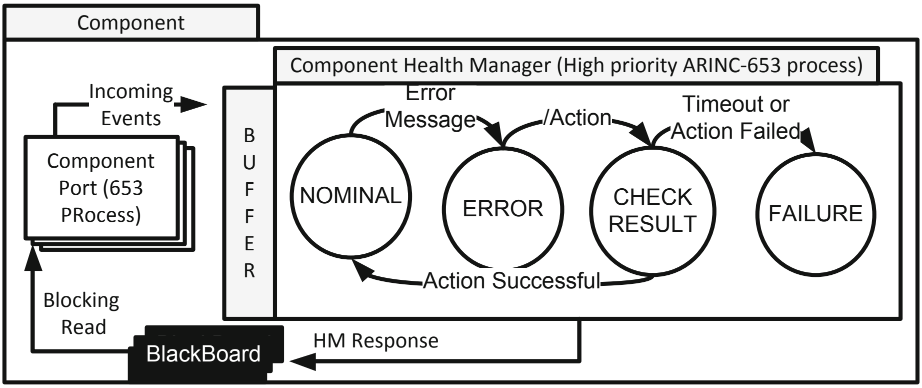


Fig. 7. Component Health Manager

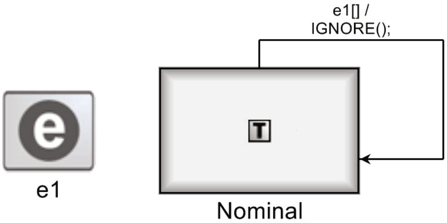


Fig. 8. Component Level Health Management Strategy for Sensor Component. Event e1 implies a user code exception in data out port.

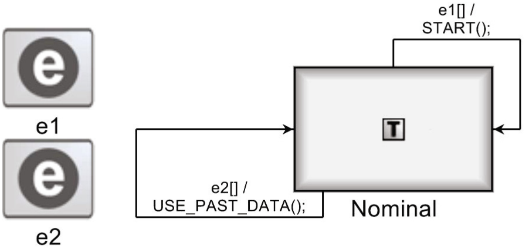


Fig. 9. Component Level Health Management Strategy for GPS Component. Event e1 implies deadline violation of data in port. Event e2 implies validity violation of data in port. Any other anomaly is sent the default IGNORE action.

USE_PAST_DATA are communicated to the managed process executing the monitor using a shared memory resource called (a "blackboard" in [1]). START, STOP and RESTART commands are directly executed using the APIs of the framework.

Example. Figure 8 shows the component health manager associated with the sensor component in the assembly shown in Figure 4. The timed-state machine specifying the CLHM for Sensor Component is triggered when a violation in the Publisher's (data_out) User-Code is detected. The monitor associated with detecting this violation is run on the same ARINC-653 process as the Publisher port. When the violation is detected, it is reported to the CLHM and the Publisher code blocks for a response/ command from the CLHM. The reported user code violation triggers the event e1 in CLHM state machine. In this case, the state-machine issues an IGNORE event which is translated as an IGNORE command and sent back to the Publisher port. Upon receipt of the command, the publisher port executes the command. In this case the IGNORE command results in the publisher continuing with its task (as per the semantics of the IGNORE command explained in the Table 2).

The timed-state machine associated with GPS CLHM is shown in Figure 9. This state machine is triggered whenever the input events e1 or e2 is triggered. The event e1 is triggered when a violation is detected in the resource usage (Deadline Violation) of the Consumer port (data_in). Event e2 is triggered when the age of the data-token received by the consumer port (data_in) is beyond its usable-time (Data-Validity Violation). The event e1 is triggered when the underlying framework detects a deadline violation and reports it to the CLHM. In this case, since the consumer port is configured with a HARD Deadline Type, the framework stops the process and then reports the violation to CLHM. This triggers the input event e1 in the state-machine. The state-machine executes the transition corresponding to the event e1 and issues a START event. This event results in a command to the framework to START the failed process associated with the consumer port. The event e2 in the CLHM state-machine is triggered when the Data-Validity violation is detected in a token received by the Consumer port. The Consumer port reports the same to the CLHM and blocks for a response from the CLHM. The CLHM executes the transition corresponding to the event e2 and triggers a USE_PAST_DATA output event which is sent as a USE_PAST_DATA command to the consumer port. The consumer port executes this command by replacing the current token with the past token and continues its operation. The Nav display machine is modeled in a similar fashion and is not shown here.

Scope of Component Level Health Manager. Inputs (anomaly detected) and outputs (commands issued) of CLHM are local to a component. While this provides a quick fix to the detected problem which could prevent the effect of the problem from being propagated, it might not solve the root-cause of the problem. In the examples discussed above, it is quite possible that an anomaly detected in one component (e.g. validity violation in GPS) could have resulted from a problem in an upstream component (e.g. Sensor's Publisher user code that is responsible for data published). Also, an anomaly observed in one component could be the effect of a CLHM mitigation action executed in another component. A higher level health management unit is required to tackle the problem of fault cascades across component boundaries. The next section deals with this second (or higher) level health management unit.

4.2 System-Level Health Manager

System Level Health Manager (SLHM), as the name suggests, is the health management strategy at the system-level. This section discusses in detail the enhancements that need to be made to the existing system made up of ACM components to enable System Level Health Management.

Architecting the Assembly Model with the SLHM Layer. Architecting support for SLHM into the existing model involves adding special components that provide the core SLHM functionality and instrumenting the existing components in the assembly with the capability to exchange information with these special components. As shown in the Figure 10, the three special SLHM components include:

– *Alarm Aggregator*: It is responsible for collecting and aggregating inputs from the component level health managers (local alarms and the corresponding mitigation actions). It hosts an aperiodic consumer that is triggered by the

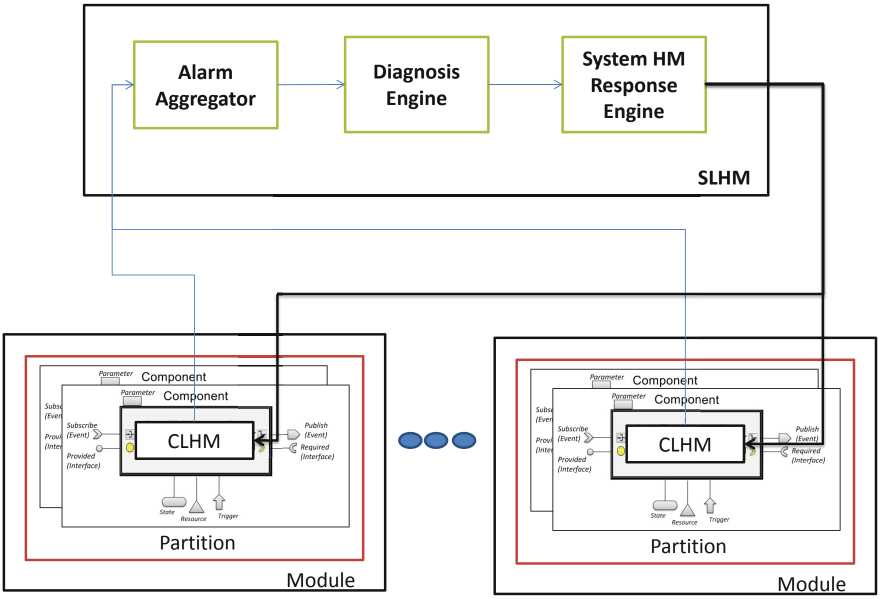


Fig. 10. SLHM Architecture. SLHM Components are automatically configured by the ACM design tools.

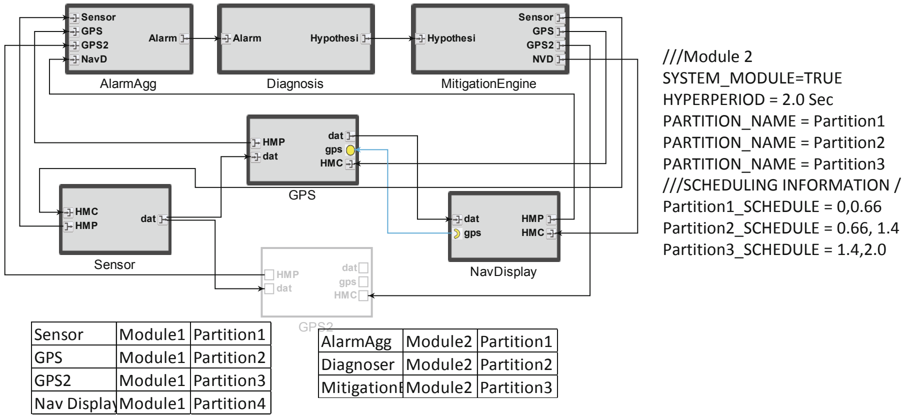
data (alarm, and local mitigation command) provided by the Component Level Health Managers. The Alarm Aggregator assimilates the information received from the CLHM-s in a moving window, whose default value is same as the hyperperiod, and sorts them based on their time of occurrence. A periodic publisher in the Alarm Aggregator feeds this sorted data to the Diagnosis Engine.

- *Diagnosis Engine*: It hosts an instance of a Timed Failure Propagation Graph reasoning engine. This engine is initialized by an auto-generated Timed Failure Propagation Graph (TFPG) model that captures the failure-modes, discrepancies and the failure propagation across the entire system. The reasoner uses this model to isolates the most plausible fault-source (component) that could explain the observations i.e. monitors triggered and the CLHM commands issued. The diagnosis result i.e. faulty component(s) is reported through an aperiodic publisher to the next component: the SystemHMResponse Engine that hosts the system level mitigation strategy.
- *SystemHM Response Engine*: It receives the diagnosis results: the set of faulty components and responds with an appropriate system-level command to mitigate the fault and its effects. This engine hosts a timed state-machine that executes the SLHM mitigation strategy specified by the user (described later in this section). The updated fault-status of the components in the assembly is used to trigger the SLHM state-machine. The output generated by the state machine is translated and sent (published) as mitigation commands to the appropriate components.

In order to enable communication between the existing component assembly and the SLHM layer, each components in the existing assembly is instrumented with an additional publisher (**HMPublisher**) and consumer (**HMConsumer**). More specifically, the special publisher (**HMPublisher**) in these components feeds CLHM output (alarm detected and local mitigation action) to the Alarm Aggregator component. The special consumer (**HMConsumer**) in these components receives the mitigation command issued by the SystemHM Response Engine and executes it.

The modeling and generator support tools automatically update the design of the entire system with the special SLHM components, additional publisher and consumer in each of the existing components, and inter-connections between the components to capture the SLHM related information flow. Two additional pieces of information are required to complete the SLHM generation - the customized mitigation strategy to be executed by the SystemHM Response Engine and the deployment information for the three SLHM components. While the deployment information can be captured in a manner similar to the other (regular/functional) components in the assembly, the design tools allow the mitigation strategy to be specified as a state machine model. The code generators use the updated model to complete the generation and customization of the SLHM layer.

Example. Figure 11 shows the GPS assembly described earlier in Figure 4 augmented with the three system health management components. Notice that each functional component i.e. Sensor, GPS, GPS2 and NavDisplay gets an additional publisher and Consumer. Anomaly/ alarms and mitigation commands are communicated through these ports. This process is completely automated. The integrator only specified the internal of the response/ mitigation engine using as a timed state machine model and specifies the SLHM deployment. In



```
///Module 2
SYSTEM_MODULE=TRUE
HYPERPERIOD = 2.0 Sec
PARTITION_NAME = Partition1
PARTITION_NAME = Partition2
PARTITION_NAME = Partition3
///SCHEDULING INFORMATION /
Partition1_SCHEDULE = 0,0.66
Partition2_SCHEDULE = 0.66, 1.4
Partition3_SCHEDULE = 1.4,2.0
```

Sensor	Module1	Partition1
GPS	Module1	Partition2
GPS2	Module1	Partition3
Nav Displa	Module1	Partition4

AlarmAgg	Module2	Partition1
Diagnoser	Module2	Partition2
MitigationE	Module2	Partition3

Fig. 11. GPS Assembly (ref, Figure 4) augmented with the SLHM components. This process is completely automated. The integrator only specified the internal of the response/ mitigation engine using as a timed state machine model and specifies the SLHM deployment.

this particular example, SLHM components are deployed on a different processor (module) and divided into three partitions. This ensures that each stage of the SLHM gets a fixed time slice. The hyper period of this module is synchronized with the hyper period of the module containing the functional components of the GPS-Assembly. This ensures that system diagnosis, mitigation and transmission of message across the two modules run synchronously.

The following sections provide more detailed information with examples on the Diagnosis and Mitigation aspects of the SLHM layer.

4.3 Diagnosis : Isolation and Identification of the Fault Source

This section focuses in more detail on the diagnosis and mitigation aspects of system health manager. Our implementation of SLHM uses a reasoning scheme based on the Timed Failure Propagation Graph (TFPG) model[3,20].Timed failure propagation graphs (TFPG) are causal models that capture the temporal characteristics of failure propagation in dynamic systems. A TFPG is a labeled directed graph. Nodes in graph represent either failure modes (fault causes), or discrepancies (off-nominal conditions that are the effects of failure modes). Edges between nodes capture the propagation of the failure effect.

The TFPG model serves as the basis for a robust online diagnosis scheme that reasons about the system failures based on the events (alarms and modes) observed in real-time[2,3,20]. The TFPG approach has been applied and evaluated for various aerospace and industrial systems[31].

Example. Figure 12 shows a simple non-hierarchical TFPG model. It shows the root causes of the failure (Failure-Modes FM_FM1, FM_FM2) and the anomalies (Discrepancies DISC_RD1, DISC_D1, DISC_SD12, DISC_D12, DISC_RD2, DISC_D2) that would be triggered when one or more of these failures occur. While failure modes are depicted as a box, unobserved OR-Discrepancies

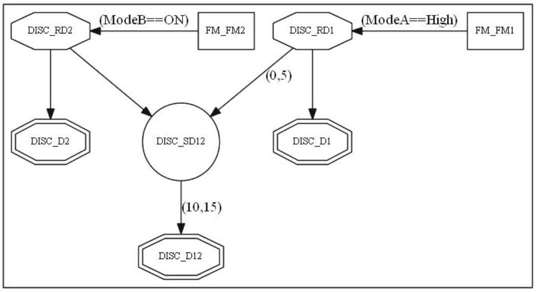


Fig. 12. An Illustrative TFPG Example. Doubled lined octagons are observed discrepancy. Single Line octagon are unobserved OR discrepancies. Unobserved AND discrepancy are denoted by circle. Rectangles are root failure nodes. Graph edge shows propagation link. Edge can be annotated with min and max propagation time. Absence of this annotation implies propagation delay lies within the interval (0,∞).

(e.g. DISC_RD1, DISC_RD2 etc.) are depicted as octagons, unobserved AND Discrepancies (DISC_SD12) are drawn as circles. Observable discrepancies (e.g. DISC_D1, DISC_D2, and DISC_D12) are drawn with a double-boundary. Edges in the graph capture the failure propagation starting from the Failure Modes to Discrepancies and then to subsequent Discrepancies downstream. Some of these links depict additional constraints related to activation-condition and timing for failure propagation. The activation condition (a Boolean expression over the modes) captures when failure can propagate over a link. The timing constraint expresses the time bounds within which the failure effect is expected to propagate over that link. When these constraints are absent, the failure can propagate over the link at any time or in any mode.

4.4 Automated Synthesis of TFPG from ACM Assembly Model

The information present in the ACM assembly model allows us to automatically synthesize the TFPG model of the system. This TFPG model is built on a hierarchical basis. Initially the TFPG models of the component ports are created. These component port TFPG models are then used to build the TFPG models of the Components which are then used to build the TFPG model of the entire Assembly.

The TFPG-model for each component-port type is constructed using the knowledge of the sequence of operation (for each port-type) and the fault-sources and anomalies associated with each operation in the sequence. The TFPG model links the fault-sources/ failure-modes and the anomalies/ discrepancies (monitored/ unmonitored) across the sequence of operations. It also contains input and output discrepancies that represent anomalies that propagate in or propagate out of the port.

The TFPG model of the component is then constructed by instantiating the appropriate component-port TFPG model for each component port present in the component. TFPG model of the component includes additional failure modes and anomalies specific to the component. The Component TFPG model is completed by adding the failure propagation links between the fault-sources and anomalies present in the component and its ports. This is done by using the data and control flow information captured in the models of the software components.

The TFPG model of an assembly is constructed using the TFPG models of the components present in the assembly. The failure propagation links between the component TFPG models are added on the basis of the integration information i.e. inter-component interaction information (publisher-consumer, facet-receptacle) present in the assembly.

The activation conditions for the failure propagation links in a component-port are expressed in terms of the mitigation commands issued by the CLHM e.g. An IGNORE command from the CLHM could imply that the failure could propagate and trigger anomalies downstream e.g. an invalid data being published or an invalid state update. An ABORT command from the CLHM could stop the failure propagation, but it also stops the normal sequence of operation of the port, thereby leading to no data being published or no update to the state. The failure propagation links across the component boundaries have activation conditions that are based on the states of the two associated components: active, inactive, semi-active. A detailed discussion of the TFPG templates associated with the port is not included in this chapter. Interested readers are referred to Appendix D in [13] and the example in [27].

4.5 Example

Figure 13 shows a portion of the TFPG model of the entire GPS Assembly. It shows the TFPG model of the sensor component, the TFPG for GPS data_in port, and the failure propagation between them.

The TFPG models of ports of components have a failure mode: FM_USER_CODE. This failure mode arises from the latent bug in implementation code. Both TFPG models also contain anomalies associated with violations observed

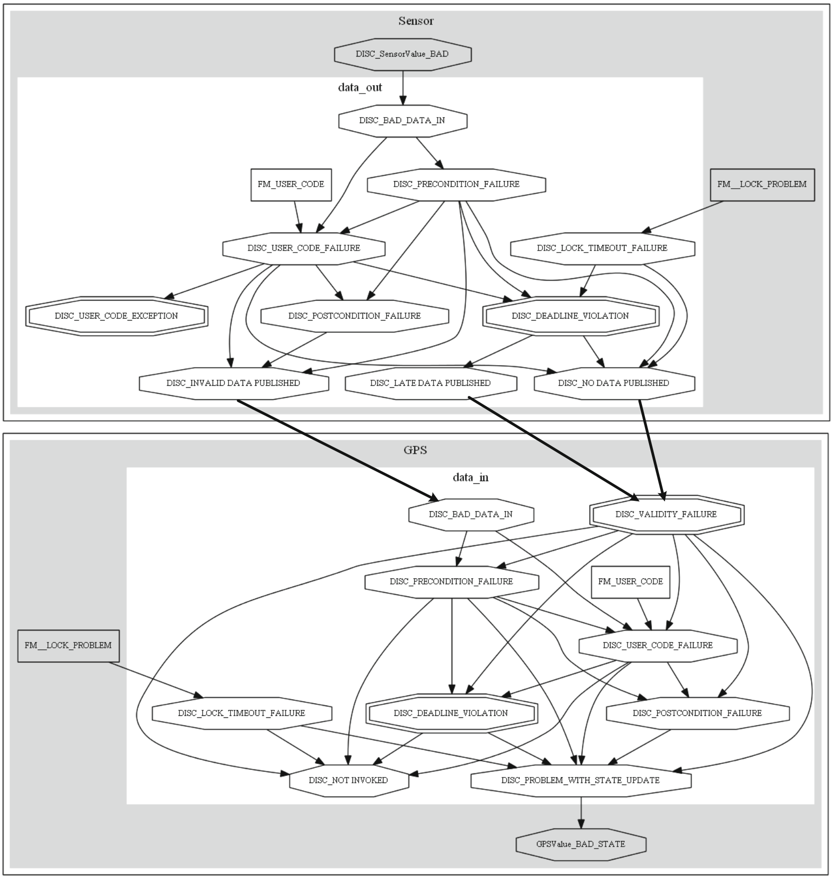


Fig. 13. TFPG model for Sensor-Publisher and GPS-Consumer

in the normal sequence of operation: data validity, pre-condition, user code exceptions, post-conditions and deadline violations. Causal relation across these anomalies is dictated by the operation sequence in the port. For example, an anomaly in the pre-condition could lead to an anomaly in the user code, which again could lead to a post-condition or deadline-violation. Latent bugs (i.e. FM_USER_CODE) can lead to an exception in the user code or can lead to deadline-violation or post-condition violation.

As stated earlier, port TFPG models here also contain input and output discrepancies that represent anomalies that propagate in or propagate out of the port. In Figure 13, DISC_BAD_DATA_IN is associated with bad-data getting into a port from a state variable (in case of a sensor-publisher) or a publisher (in case of a GPS-consumer). DISC_LOCK_TIMEOUT_FAILURE represents another input-discrepancy that is associated with inability to secure the component-lock. Anomalies propagating out of the publisher port are captured by discrepancies associated with the published data, e.g. DISC_NO_DATA_PUBLISHED, DISC_LATE_DATA_PUBLISHED, and DISC_INVALID_DATA_PUBLISHED. Similarly, anomalies out of consumer port are captured by discrepancies that are associated with problem in the state-update , e.g. DISC_PROBLEM_WITH_ STATE_UPDATE.

Activation conditions on failure propagation links are not shown in Figure 13. These conditions are based on local mitigation commands. For example, an IGNORE or USE_PAST_DATA command from CLHM in response to a violation detected in a pre-condition can cause problems in the user code or post-condition, or deadline violation. Finally, this could lead to a bad output data. On the other hand, an ABORT command can arrest the failure propagation in nominal operation sequence but could introduce other effects such as no output data (e.g DISC_NO_DATA_PUBLISHED).

Figure 13, also shows the component wide failure modes such as FM_LOCK_PROBLEM. This failure mode represents the problem associated with synchronization among component ports. The effect of this failure manifests in a component port through the discrepancy: DISC_LOCK_TIMEOUT_FAILURE. Component TFPG models also contain anomalies associated with bad values in the state variables: DISC_Sensor_Bad_Value and GPSValue_Bad_State. These anomalies help in capturing the failure propagation based on the dataflow model of the component. The bad data produced in a port could lead to a bad state variable update in a component. If the state variable is being used by a publisher port for publishing data, the bad state variable update can lead to an invalid data being published from the publisher port. This failure propagation associated with dataflow is not restricted to the component boundary.

Dataflow due to the component port interactions captured in the assembly model could lead to failure propagations across component boundaries. Figure 13 captures these failure propagations across component boundaries between the sensor' publisher port to the GPS's consumer port (dark edges in the Figure 13). These failure propagations capture the effect of problems in the sensor's publisher trickling down

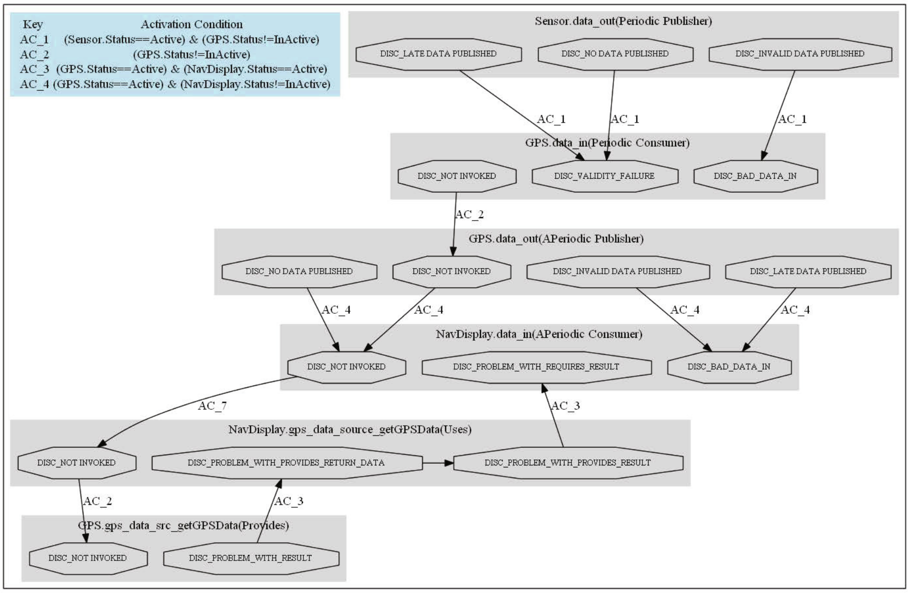

Key
Activation Condition
AC_1 (Sensor.Status==Active) & (GPS.Status!=InActive)
AC_2 (GPS.Status!=InActive)
AC_3 (GPS.Status==Active) & (NavDisplay.Status==Active)
AC_4 (GPS.Status==Active) & (NavDisplay.Status!=InActive)
Sensor.data_out(Periodic Publisher)
DISC_LATE DATA PUBLISHED
DISC_NO DATA PUBLISHED
DISC_INVALID DATA PUBLISHED
AC_1
AC_1
AC_1
GPS.data_in(Periodic Consumer)
DISC_NOT INVOKED
DISC_VALIDITY_FAILURE
DISC_BAD_DATA_IN
AC_2
GPS.data_out(APeriodic Publisher)
DISC_NO DATA PUBLISHED
DISC_NOT INVOKED
DISC_INVALID DATA PUBLISHED
DISC_LATE DATA PUBLISHED
AC_4
AC_4
AC_4
AC_4
NavDisplay.data_in(APeriodic Consumer)
DISC_NOT INVOKED
DISC_PROBLEM_WITH_REQUIRES_RESULT
DISC_BAD_DATA_IN
AC_7
AC_3
NavDisplay.gps_data_source_getGPSData(Uses)
DISC_NOT INVOKED
DISC_PROBLEM_WITH_PROVIDES_RETURN_DATA
DISC_PROBLEM_WITH_PROVIDES_RESULT
AC_2
AC_3
GPS.gps_data_src_getGPSData(Provides)
DISC_NOT INVOKED
DISC_PROBLEM_WITH_RESULT

Fig. 14. Intra-component and Inter-component Failure Propagation associated with the control-flow in the GPS Assembly

into GPS's consumer leading to a bad input data (DISC_BAD_DATA_IN) or a data validity violation (DISC_DATA_VALIDITY).

Failure effects are also propagated along the control flow, e.g. when a port is not invoked: DISC_NOT_INVOKED. This happens as the control flow is disrupted when the normal sequence of operation is affected in a port. This can happen across component boundary if a facet-receptacle interaction exists or within the component when a port is responsible for invoking another port, e.g. a periodic consumer can invoke an aperiodic publisher. In Figure 14, this relationship exists between GPS data_in and GPS data_out. A lack of invocation of a port can affect the state update inside the component.

Figure 14 captures the explicit failure propagations across component port boundaries. These failure propagations include those introduced by dataflow as well as control flow. To avoid clutter, the Figure 14 restricts the depiction to anomalies within component ports that are associated with direct failure propagations across component-port or component boundaries. Other failure modes, anomalies, and failure propagations within component port boundaries are not shown.

4.6 System Level Diagnosis Process

The TFPG diagnosis engine hosted inside the SHM component is instantiated with the generated TFPG model of the system/assembly. When it receives the first alarm from a fault scenario, it reasons about it by generating all hypotheses for failure modes that could have possibly triggered the alarm. Each hypothesis lists its possible failure modes and their possible timing interval, the triggered alarms that are supportive of the hypothesis, the triggered alarms that are inconsistent with the hypothesis, the missing alarms that should have triggered, and the alarms that are expected to trigger in future. Additionally, the reasoner computes hypothesis metrics such as plausibility and robustness that provide a means of comparison. These metrics are used to prune the hypotheses set such that only those hypotheses with a higher metric and hence better explanation are retained [2]. At this time only hypothesis with 100 percent plausibility are used for failure mitigation. Output of diagnosis engine i.e. the hypothesis of failed component is sent to the timed state machine implementing the System Level Mitigation Strategy.

Example. Consider a failure scenario such that there is a user code bug in the sensor data_out process such that the code does not publish the data in its time duration. Since sensor data out process does not have a monitored post-condition discrepancy (see Figure 13) there will be no alarms generated in the sensor process. However, due to the user code problem, the silent (unobserved) discrepancy user code failure will be triggered, which will then lead to either silent post-condition failure, or late data published, or no data published.

Now, consider the GPS data in process in the same figure. In this process, a validity violation will be raised as no data is being received from the publisher. This will cause the local health manager to issue a USE_PAST_DATA command (Figure 9). The raised alarm of validity violation along with the USE_PAST_DATA command will be reported to the diagnosis engine. Inside the diagnosis engine, the event of 'validity violation' will be used to produce the most plausible hypothesis (root failure sources) that can explain the observed anomaly. In this particular case the TFPG engine will correctly attribute the problem to either Sensor Lock failure mode or Sensor user code failure mode, i.e. a faulty sensor component.

4.7 System Level Mitigation Strategy

The system level mitigation strategy is also modeled as a hierarchical timed state machine. Table 3 lists the statements (functions) that can be used in the state machine to express the guard conditions (to check if a component is faulty) and actions (i.e. mitigation commands). These strategies are reactive in nature and aim to restore the functionality by cold/warm reset or switching to redundant component. As mentioned earlier in this chapter, each component in the assembly is assumed to be in one of the three possible states: inactive, active, and semi-active. When the component is in inactive state, none of the ports in the component perform their task. The active state of a component is the exact opposite to inactive state, and all the component ports performing their task. In a semi-active state, only the consumer and receptacle ports of a component are operational. The publisher and provided ports are disabled. This state-machine

Table 3. SLHM Functions. Here c denotes the component name and s denotes a subsystem name. Unless otherwise specified usage of the subsystem name in a command implies apply to all contained components.

Action	**Semantics**
IS_FAULTY (c|s)	Returns true if the component is faulty.. A subsystem is marked as faulty if the minimum number of components required for work is not available.
IS_NOT_FAULTY (c)	Returns false if the component is faulty.[1]
RESET (c|s)	Instructs the component to execute its Reset method.
STOP (c|s)	Instructs the component to switch to inactive mode. Component stops executing the functionality of all its ports. If subsystem is argument, command is applied to all its components
START (c|s)	Instructs the component to switch to active mode. Component starts executing the functionality of all its ports.
DISABLE_OUTPUT (c|s)	Instructs the component to switch to semi-active mode. Only Consumer and Receptacle ports are operational.
REWIRE (c,i,pc)	i: Interface Name, pc: Provider Component Name. This command Instructs Component (c) to switch its receptacle Interface (i) to connect to the appropriate facet interface in another component (pc).
CHECKPOINT (c|s)	Instructs the component to Checkpoint its current state-variables.
RESTORE (c|s)	Instructs the component to Restore its state-variables from the Checkpoint.

[1] A corresponding method can be implemented for the subsystem and used in a specific example. Currently, implementation of this method is system specific and is not part of provided API.

is translated into operational code and is hosted inside the runtime of the System Level Health Management module.

An alternative strategy of health management is to search over available solutions to find the best option that can ensure that the system functionalities are still met. This strategy is still under active investigation.

Example. Figure 15 shows the state-machine model of the System Level Mitigation Strategy associated with the GPS Assembly. In this case the mitigation strategy involves parallel state machines that deal with problems associated with Sensor component (parallel-state 1) and GPS component (parallel-state 2). The top level state machine is triggered when there is updated diagnosis information (hypothesis) from the diagnosis engine. This information is used by the System response/ mitigation engine to update the list of faulty components and trigger the SLHM state machine. When the SLHM state machine (Figure 15) associated with the GPS Assembly is invoked it triggers Parallel State-1 followed by Parallel State-2.

In Parallel-State 1, the System Health Manager checks if the GPS component is marked as faulty. This is captured in the transition guard condition:

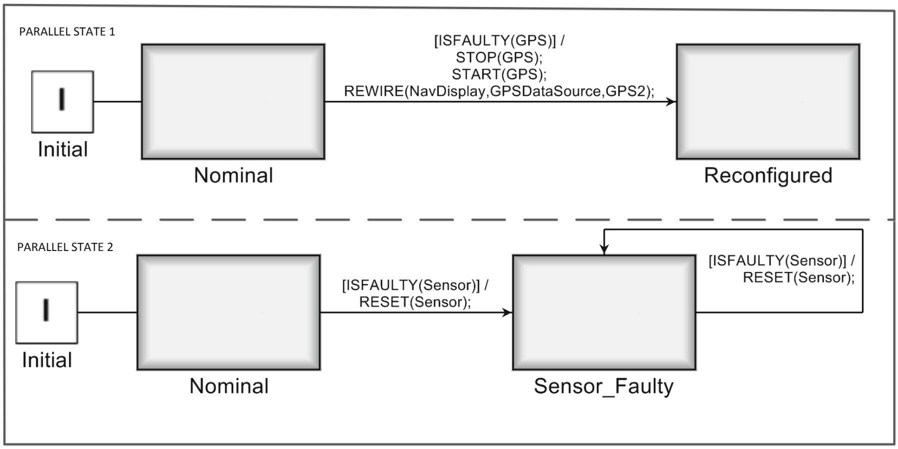


Fig. 15. System Level Health Manager Mitigation Strategy for GPS Assembly

IS_FAULTY(GPS). If this guard condition evaluates to true, then the transition action statements are executed and the state changed. The transition action statements direct the reconfiguration to the alternate GPS (GPS2) through a series of statements. STOP(GPS) results in a STOP command being sent to the GPS component. START(GPS2) results in a START command being sent to the GPS2 component. The command REWIRE(NavDisplay,GPSDataSource, GPS2) directs a rewire command to the NavDisplay component. It instructs the component to rewire the receptacle interface (GPSDataSource) to the appropriate facet in the component GPS2.

In Parallel-State 2 it checks if the Sensor component has been marked as faulty. This is captured in the transition guard condition: IS_FAULTY(Sensor). If this guard condition evaluates to true, then an output event is triggered to reset the sensor component (transition action: RESET(Sensor)). This translates into a RESET command that is sent to the Sensor component. The Sensor component then executes the Reset method associated with the component and reports back to the system health manager.

An additional case study of an Inertial Measurement Unit (IMU)System built using the ACM design tools is available as a tech report for interested readers [16].

5 Known Limitations and Future Work

While the results of the experiments indicate that the approach is feasible and very general, and shows the promise of being able to scale to and handle real-life problems, we do understand that architecting a software health management system is contingent upon the availability of extra resources that can be spared for this purpose. This implies the necessity of scheduling analysis that considers the future state change of components as part of system mitigation. On the modeling front, the current state-machine based mitigation strategy requires explicit specification of the mitigation action for each fault in the system.

This might become unwieldy beyond a point. We are focusing on alternate strategies to specify the mitigation action. We are exploring the use of function-allocation models in conjunction with automated reasoning strategies to tackle the mitigation problem by identify and switching to available redundancies to restore the affected functionality. Further, we need to explore effective means to use the diagnosis result when it is less than perfect i.e. when the hypotheses are not good enough to accurately determine the faulty component. On the diagnosis front, it would be ideal if all the possible monitors (i.e. pre-conditions, post-conditions, invariants) are configured and available for use with the TFPG diagnosis model. In an ideal monitoring situation where all monitors are configured correctly and fire in the correct sequence, this will help prune the hypotheses set faster and come up with a quick and correct diagnosis result. However, we do understand that it might not be possible to configure all the available monitors and more importantly and in some cases these monitors could not be reliable. We plan to work on strategies where less than perfect diagnosis results (lots of ambiguities and/or lack of hypotheses that have hundred percent plausibility) can be effectively handled to restore the system functionality.

6 Conclusion

In summary, the paper describes a technology for implementing fault adaptivity in real-time systems using as software health management approach. The starting point of the technology is a real-time component model that introduces component-based software engineering techniques into real-time systems. Components, their interfaces, and interactions are explicitly modeled, and these models are annotated with observable pre- and post-conditions, as well as timing requirements. An anomaly detection system is constructed from these specifications. It performs the monitoring on the software system, and, if needed, triggers a health management (mitigation) action. Health management can happen on the component or on the system-level: in the first case the mitigation is facilitated by a designer-specified reactive state machine, in the second case a diagnosis process is triggered first, whose results are then used in a reactive or deliberative response/ mitigation engine. The diagnosis is necessary to isolate source of cascading faults that propagate through multiple components. We have built a model-driven tool chain for developing these systems, and we have evaluated the approach on several laboratory examples and demonstrated the effectiveness of the concept on some large ones that replicate real-life incidents.

References

1. ARINC specification 653-2: Avionics application software standard interface part 1 - Required Services. Aeronautical Radio, lnc.
2. Abdelwahed, S., Karsai, G., Mahadevan, N., Ofsthun, S.C.: Practical considerations in systems diagnosis using timed failure propagation graph models. IEEE Transactions on Instrumentation and Measurement 58(2), 240–247 (2009)

3. Abdelwahed, S., Karsai, G., Biswas, G.: A consistency-based robust diagnosis approach for temporal causal systems. In: 16th International Workshop on Principles of Diagnosis, pp. 73–79 (2005)
4. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004)
5. Bureau, A.T.S.: In-flight upset; 240km NW Perth, WA; Boeing Co 777-200, 9M-MRG. Tech. rep. (August 2005), http://www.atsb.gov.au/publications/investigation_reports/2005/AAIR/aair200503722.aspx
6. Bureau, A.T.S.: AO-2008-070: In-flight upset, 154 km west of Learmonth, WA, 7, VH-QPA, Airbus A330-303. Tech. rep (October 2008), http://www.atsb.gov.au/publications/investigation_reports/2008/AAIR/aair200806143.aspx
7. Bustard, D.W., Sterritt, R.: A requirements engineering perspective on autonomic systems development. In: Autonomic Computing: Concepts, Infrastructure, and Applications, pp. 19–33 (2006)
8. Butler, R.: A primer on architectural level fault tolerance. Tech. rep., NASA Scientific and Technical Information (STI) Program Office, Report No. NASA/TM-2008-215108 (2008), http://shemesh.larc.nasa.gov/fm/papers/Butler-TM-2008-215108-Primer-FT.pdf
9. Charette, R.: This car runs on code. IEEE Spectrum (February 2009)
10. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: 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)
11. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: Towards architecture-based self-healing systems. In: WOSS 2002: Proceedings of the First Workshop on Self-healing Systems, pp. 21–26. ACM Press, New York (2002)
12. DO-178B, Software considerations in airborne systems and equipment certification. RTCA, Incorporated (1992)
13. Dubey, A., Karsai, G., Mahadevan, N.: Towards model-based software health management for real-time systems. Tech. Rep. ISIS-10-106, Institute for Software Integrated Systems, Vanderbilt University (August 2010), http://isis.vanderbilt.edu/node/4196
14. Dubey, A., Karsai, G., Mahadevan, N.: A component model for hard real-time systems: CCM with ARINC-653. Software: Practice and Experience 41(12), 1517–1550 (2011), http://dx.doi.org/10.1002/spe.1083
15. Dubey, A., Karsai, G., Mahadevan, N.: Model-based Software Health Management for Real-Time Systems. In: IEEE Aerospace Conference, pp. 1–18. IEEE (2011)
16. Dubey, A., Mahadevan, N., Karsai, G.: The inertial measurement unit example: A software health management case study. Tech. Rep. ISIS-12-101, Institute for Software Integrated Systems, Vanderbilt University (February 2012), http://isis.vanderbilt.edu/node/4496
17. Garlan, D., Cheng, S.W., Schmerl, B.: Increasing System Dependability Through Architecture-based self-repair. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems. LNCS, vol. 2677, pp. 61–89. Springer, Heidelberg (2003), http://dl.acm.org/citation.cfm?id=1768179.1768183

18. Greenwell, W.S., Knight, J., Knight, J.C.: What should aviation safety incidents teach us? In: SAFECOMP 2003, The 22nd International Conference on Computer Safety, Reliability and Security (2003)
19. Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming 8(3), 231–274 (1987), http://www.sciencedirect.com/science/article/pii/0167642387900359
20. Hayden, S., Oza, N., Mah, R., Mackey, R., Narasimhan, S., Karsai, G., Poll, S., Deb, S., Shirley, M.: Diagnostic technology evaluation report for on-board crew launch vehicle. Tech. rep., NASA (2006)
21. Jaffe, M., Busser, R., Daniels, D., Delseny, H., Romanski, G.: Progress report on some proposed upgrades to the conceptual underpinnings of do-178b/ed-12b. In: 2008 3rd IET International Conference on System Safety, pp. 1–6. IET (2008)
22. Johnson, S., Gormley, T., Kessler, S., Mott, C., Patterson-Hine, A., Reichard, K., Scandura Jr., P.: System Health Management: With Aerospace Applications. John Wiley & Sons, Inc. (2011)
23. de Lemos, R.: Analysing failure behaviours in component interaction. Journal of Systems and Software 71(1-2), 97–115 (2004)
24. Lightstone, S.: Seven software engineering principles for autonomic computing development. ISSE 3(1), 71–74 (2007)
25. Lyu, M.R.: Software Fault Tolerance. John Wiley & Sons, Inc., New York (1995), http://www.cse.cuhk.edu.hk/~lyu/book/sft/
26. Lyu, M.R.: Software reliability engineering: A roadmap. In: 2007 Future of Software Engineering, FOSE 2007, pp. 153–170. IEEE Computer Society, Washington, DC (2007), http://dx.doi.org/10.1109/FOSE.2007.24
27. Mahadevan, N., Dubey, A., Karsai, G.: Application of software health management techniques. In: Proceedings of the 2011 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011. ACM, New York (2011)
28. Potocti de Montalk, J.: Computer software in civil aircraft. In: IEEE/AIAA 10th Digital Avionics Systems Conference, pp. 324–330 (October 1991)
29. NASA: Report on the loss of the mars polar lander and deep space 2 missions. Tech. rep., NASA (2000), ftp://ftp.hq.nasa.gov/pub/pao/reports/2000/2000_mpl_report_1.pdf
30. Ofsthun, S.: Integrated vehicle health management for aerospace platforms. IEEE Instrumentation Measurement Magazine 5(3), 21–24 (2002)
31. Ofsthun, S.C., Abdelwahed, S.: Practical applications of timed failure propagation graphs for vehicle diagnosis. In: Proc. IEEE Autotestcon, September 17-20, pp. 250–259 (2007)
32. Prisaznuk, P.: Arinc 653 role in integrated modular avionics (IMA). In: IEEE/AIAA 27th Digital Avionics Systems Conference, DASC 2008, pp. 1.E.5–1 – 1.E.5–10. IEEE (2008)
33. Pullum, L.L.: Software fault tolerance techniques and implementation. Artech House, Inc., Norwood (2001)
34. Robertson, P., Williams, B.: Automatic recovery from software failure. Commun. ACM 49(3), 41–47 (2006)
35. Rohr, M., Boskovic, M., Giesecke, S., Hasselbring, W.: Model-driven development of self-managing software systems. In: Proceedings of the Workshop "Models@run.time" at the 9th International Conference on model Driven Engineering Languages and Systems, MoDELS/UML 2006 (2006)
36. Sha, L.: The complexity challenge in modern avionics software. In: National Workshop on Aviation Software Systems: Design for Certifiably Dependable Systems (2006)

37. Shaw, M.: "self-healing": softening precision to avoid brittleness: position paper for woss 2002: workshop on self-healing systems. In: WOSS 2002: Proceedings of the First Workshop on Self-healing Systems, pp. 111–114. ACM Press, New York (2002)
38. Srivastava, A., Schumann, J.: The Case for Software Health Management. In: Fourth IEEE International Conference on Space Mission Challenges for Information Technology, SMC-IT 2011, pp. 3–9 (August 2011)
39. Taleb-Bendiab, A., Bustard, D.W., Sterritt, R., Laws, A.G., Keenan, F.: Model-based self-managing systems engineering. In: DEXA Workshops, pp. 155–159 (2005)
40. Torres-Pomales, W.: Software fault tolerance: A tutorial. Tech. rep., NASA (2000), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.8307
41. Wallace, M.: Modular architectural representation and analysis of fault propagation and transformation. Electron. Notes Theor. Comput. Sci. 141(3), 53–71 (2005)
42. Wang, N., Schmidt, D.C., O'Ryan, C.: Overview of the CORBA component model. In: Component-based Software Engineering: Putting the Pieces Together, pp. 557–571 (2001)
43. Williams, B., Ingham, M., Chung, S., Elliott, P.: Model-based programming of intelligent embedded systems and robotic space explorers. Proceedings of the IEEE 91(1), 212–237 (2003)
44. Williams, B.C., Ingham, M., Chung, S., Elliott, P., Hofbaur, M., Sullivan, G.T.: Model-based programming of fault-aware systems. AI Magazine 24(4), 61–75 (2004)
45. Zhang, J., Cheng, B.H.C.: Specifying adaptation semantics. In: WADS 2005: Proceedings of the 2005 Workshop on Architecting Dependable Systems, pp. 1–7. ACM, New York (2005)
46. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: ICSE 2006: Proceeding of the 28th International Conference on Software Engineering, pp. 371–380. ACM, New York (2006)

Towards User Tailoring of Self-Adaptation in Ubiquitous Computing

João Pedro Sousa

Department of Computer Science, George Mason University,
Fairfax, VA 22030, USA
jpsousa@cs.gmu.edu

Abstract. Ubiquitous computing both emphasizes the role of self-adaptation and poses new challenges to self-adaptation. These challenges include the need for *new kinds* of adaptation and the recast of classic ones, namely self-healing, to fit ubiquitous computing environments. Furthermore, because users will play an increasing role in assembling ubicomp systems, a key question is how to enable users to tailor self-adaptation to their needs.

To position the new kinds of self-adaptation, the paper proposes a classification of self-adaptation according to what gets changed in response to what, complementarily to a classification of control loops. Specifically, the paper introduces *design meshing*, concerning dynamic adaptation to requirements independently put forth by multiple users; *pliable apps*, concerning structural modes of operation in response to context or other events; and a decentralized, lightweight protocol for *self-healing*. In addition to semantic aspects, we propose language constructs for users to tailor these kinds of self-adaptation. The paper discusses a decentralized approach to implement these building on common principles such as service orientation and the ability to automatically deploy system models.

Keywords: self-adaptive software, domain-specific languages, end-user design, ubiquitous computing.

1 Introduction

The boundaries of design and run time in the software lifecycle are in a state of flux. The push for self-healing and self-adaptive systems propelled *service discovery* towards run time: the decision of which component to invoke used to be made manually during design in component-based software and early service-oriented systems [1], but can now be made automatically at run time [2], [3].

Paradoxically, that shift of responsibility from design to run time created the need to *expand design*, enriching it with specifications of service types and constraints on quality of service (QoS). If design captures these decisions in a machine-interpretable form, the subsequent human labor to *translate* the decisions into code and ultimately into the running system may be substantially reduced or even eliminated by automation (e.g., [4–6]). In either case, the creation of machine-interpretable models

R. de Lemos et al. (Eds.): Self-Adaptive Systems, LNCS 7475, pp. 324–353, 2013.

at *run time*, such as service ontologies and registries, models to estimate delivered QoS, etc., enable the automation of service discovery mechanisms. These mechanisms eliminate human labor at run time for discovering and binding services each time the system needs to adapt to failures or to lack of performance.

This paper focuses on an additional shift of responsibilities that results from the push towards ubiquitous computing (aka *ubicomp*, or pervasive computing). In addition to the use of mobile devices, ubicomp encompasses devices embedded or scattered in spaces ranging from homes to subway stations to streets and farms. Its application domains include health care and assisted living, search and rescue, disaster response, energy management in buildings, safety, transportation, social computing, home automation and surveillance, etc.

The paper argues that ubicomp both emphasizes the role of self-adaptation and poses some new challenges *to* self-adaptation. As before, new responsibilities at run time mean that more needs to be designed. The paper also argues that domain experts and end users will play an increasing role in designing ubicomp systems; and therefore, a key question is how to enable users to tailor self-adaptation to their needs.

In the remainder of this paper, section 2 elaborates on the role of self-adaptation in ubicomp, and section 3 introduces a classification of different kinds of self-adaptation to help position the kinds proposed here. Section 4 summarizes a state-of-art design notation for ubicomp that serves as a basis for the examples throughout the paper.

Sections 5, 6, and 7 respectively describe the new kinds of adaptation: *design meshing*, *pliable apps*, and B*self-healing*. Section 8 discusses their implementation and commonalities, and section 9 compares with related work. Section 10 summarizes the main points of the paper and future work.

2 Motivation

Ubicomp offers fertile ground for self-adaptation, since repairs and tune-ups are called for by several kinds of stimuli. First, ubicomp systems are often deployed in open and highly dynamic environments. Unlike conventional software systems, where stakeholders are expected to agree on a set of requirements before a system is built, ubicomp users bring their own independent requirements to the spaces they visit.

From the point of view of a space and of the systems therein, *requirements change* as users come and go. For example, a system in a smart home may open at least some of its features to impromptu users and their devices in cases such as friends coming over for a party, a nurse visiting an elderly person, or firemen arriving at a scene. With purely software systems, such scenarios result in the dynamic deployment of *separate* systems for each set of requirements. In ubicomp, the unavoidable sharing of cyber-physical services, such as thermostats, leads instead to *one* evolving system.

Second, application requirements may call for dynamic change of features in response to context, aka *context awareness*, corresponding to modes of operation in different situations. Some of these changes may be parametric, e.g. decide on the ring volume of a cell phone depending on the location, but other changes may involve significant reconfiguration, e.g., of a building automation system during a fire.

And third, ubicomp environments are often harsh on software systems: users may arbitrarily turn off devices, forget to change batteries, or disconnect cables while moving around in their activities. The challenge becomes to recover from the failure of *any* component, including those with responsibility for self-healing.

Satisfactory solutions for self-adaptation in ubicomp may have to address some combination of the kinds of stimuli above, as required by each application domain.

As happened with service discovery, additional responsibilities for the system at run time imply an expansion of design, this time to capture decisions concerning adaption to these kinds of stimuli.

Design notations that target end users and domain experts will play an important role in capturing these decisions. End users are frequently the main operators of ubicomp systems, taking up significant responsibilities in their configuration, deployment, and maintenance [7–9]. This is because, in many domains, system requirements are too personalized, too specific to circumstances, and may change too often to make the approach of hiring engineers to make every change economically feasible or even fast enough to be useful.

End-user notations must find a balance between being powerful enough to be *useful*, and being simple enough to be *usable*. Such notations will not replace generic design notations and software development frameworks for professional engineers [10], but may have an important role in democratizing access to ubicomp, similarly to the role spreadsheets had in democratizing access to personal computing in the 80's.

The contributions of this paper include language constructs for end users to design ubicomp self-adaptation to the three kinds of stimuli identified above. Specifically, *design meshing* responds to requirements independently put forth by multiple users, *pliable* a*pps* respond to context raised to application-level events, and B*self-healing* responds to failures in a resilient, decentralized fashion. The proposed constructs target end users and carry enough semantics to make them machine-interpretable.

The paper also describes the automated translation of such design decisions into models at run time, and the mechanisms that interpret those models to control ubicomp self-adaptation. Thanks to this automation, the role of humans shifts from operational to strategic: human stakeholders design adaptation policies and systems translate the policies and perform all necessary adaptations autonomously at run time.

3 Classifying Self-Adaptation

Self-adaptation refers to a system's ability to change some aspect of itself, at runtime. To help position the contributions herein, we start by characterizing self-adaptation in terms of what *kinds of changes* are made in response to what kinds of stimuli.

Table 1 distinguishes three kinds of changes: to the code base, to the run-time structure, and to the behavior of the system. For example, cell B2 corresponds to the ability to change a system's run-time structure, e.g. by adding/removing architectural components and connectors, in response to failures. Cell B3 refers to a similar ability, but in response to run-time measurements of system resources and performance. Classic work in self-adaptation, such as Rainbow [11], covers cells B2-3.

Table 1. Adaptation of different subjects in response to different stimuli

	change: in response to:	A computation / behavior	B run-time structure	C code base
	1 environment metrics	context-aware phone ring		
	2 system failures		***B*self-healing** self-healing	
	3 system metrics		self-optimizing	
	4 application events		**pliable apps**	self-modifying code
explicit user control	5 parameters			
	6 QoS goals			
	7 design artifacts	**meshing** (resolution)	**meshing**	
	8 functional requirements			automated MDD

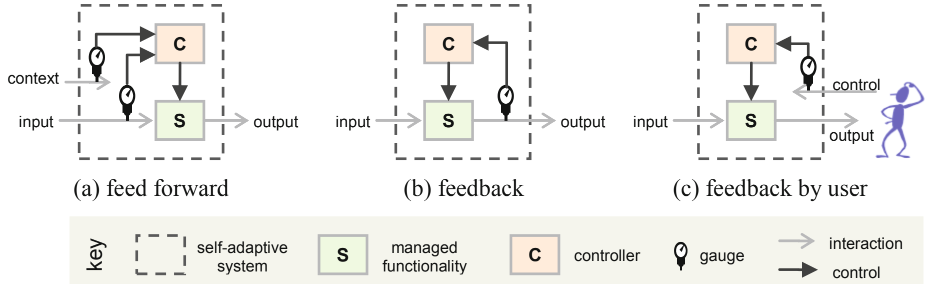
Self-adaptation is not something that occurs once, e.g. during system deployment: it occurs in response to, and every time an entity of interest changes. For example, if Quality of Service (QoS) goals are factored in during system development and then used to guide QoS optimization at run time, then the system may be adaptive to performance metrics, B3, but not to QoS goals, B6. For that, it would need to monitor changes to QoS goals and autonomously change its management of QoS optimization in response to changes in QoS goals.

The table offers a broad view of self-adaptation. For example, a cell phone performs A1 self-adaptation if it senses context, such as user location and social activity, e.g. in a meeting, and adjusts the ringing volume accordingly. On the opposite end of the table, an automated Model-Driven Development (MDD) tool performs C8 self-adaptation if it monitors functional requirements and changes the code base accordingly. Again, it is not enough to run the cycle between requirements and code once to qualify as self-adaptation: changes in requirements would need to be autonomously monitored and reacted to at run-time.

Of the kinds of self-adaptation covered in this paper, design meshing sits in A-B7. Structural adaptation, B7, results from users putting forth at run time design artifacts specifying desired features. Behavior adaptation, A7, results from the *resolution* of conflicts that may arise when multiple users compete for the same feature. Pliable apps sits in B4, since it makes changes to the run-time structure in reaction to application-level events, and *B*self-healing recasts the classic B2 for ubicomp.

This classification is orthogonal to the choice of *mechanisms* to realize self-adaptation. For example, self-adaptation may be performed by the application logic or it may be factored out to a separate controller; it may include sophisticated planning of changes [12] or it may run prepared scripts [11]; it may be distributed [13], or it may reside in dedicated central components [11], [14].

The classification in Table 1 is also distinct from a control theory classification: Figure 1. Self-adaptation has a *feed forward loop* if it reacts to independent variables,



(a) feed forward (b) feedback (c) feedback by user

Fig. 1. Control-theory classification of control loops

and has a *feedback loop* if it reacts to dependent variables. For example, a mechanism that adapts the number of replicas of a web server is B3 feed forward if it gauges and reacts to the volume of requests, an independent variable, but it is B3 feedback if it gauges and reacts to average response time, which depends on the run-time structure.

Control loops may be *implicitly* closed by users: they observe the output of the system and if not happy with some aspect of quality they may change their own behaviors. In the example above of the feed forward adaptive web server, users might react to a significant increase in response time by backing off, maybe intending to try again later, thereby reducing the volume of requests. *Mechanism design* is a field of economics that studies control loops implicitly closed by humans and proposes reward/punishment mechanisms to influence the behavior of users [15].

Lines 5 to 8 in Table 1 pertain to loops *explicitly* closed by users: Figure 1(c). These loops monitor explicit user controls, such as control parameters and QoS goals, and affect changes to the behavior, structure, or code base of the system. For example, line 7 pertains to systems that monitor *design artifacts* that are amenable to automated (re)deployment should any changes be made by users at run time.

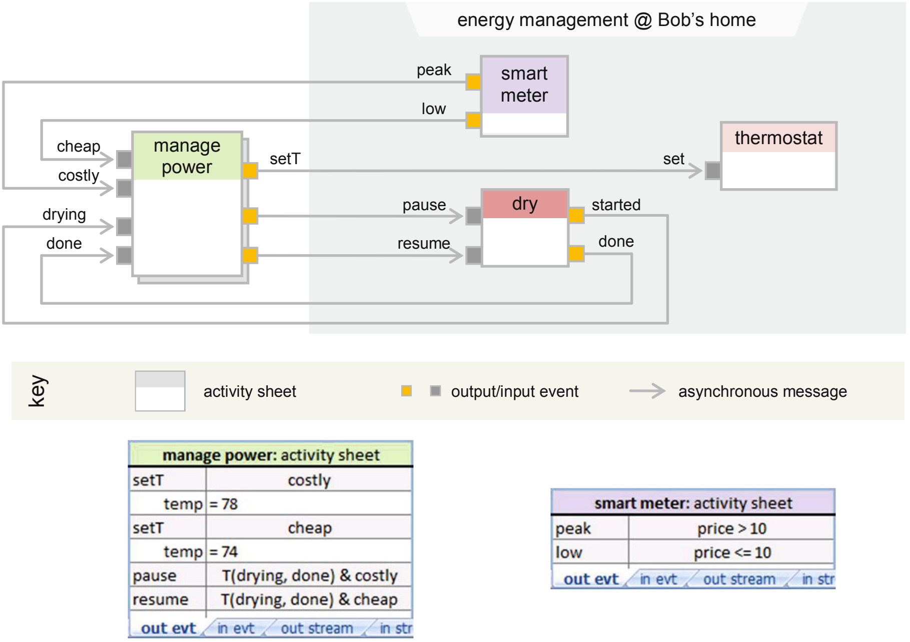
While the loops in Figure 1(a-b) adequately address *anticipated* variations with pre-packaged strategies, the loop in Figure 1(c) addresses *unanticipated* situations by allowing users to carry out new adaptations at run time.

Similarly to architectural styles, where real systems rarely conform to a single style [16], self-adaptive systems will frequently combine several kinds of adaptation loops. For example, while *B*self-healing supports a feedback loop, both design meshing and pliable apps combine automated (feed forward/feedback) loops with the users' ability to explicitly close an additional control loop: more below.

4 End-User Design of Ubicomp Systems

For concreteness, the examples throughout this paper build on a language developed in prior work, TeC, which is summarized here while the following sections focus on its extensions towards user-tailored self-adaptation.

TeC is meant for a range of end users, from home owners to domain experts such as facility administrators and health-care professionals.



manage power: activity sheet	
setT	costly
temp	= 78
setT	cheap
temp	= 74
pause	T(drying, done) & costly
resume	T(drying, done) & cheap

out evt / in evt / out stream / in str

smart meter: activity sheet	
peak	price > 10
low	price <= 10

out evt / in evt / out stream / in str

Fig. 2. Example team design, where signals from the smart electric meter at Bob's home are used to pause the clothes dryer and to adjust the AC thermostat

To become accessible to end users, TeC departs from an algorithmic model of computing in favour of a declarative model similar to spreadsheets. In the latter, there is no algorithm or "main" program: all formulas are asynchronously recalculated whenever the values they refer to are updated and the overall calculation emerges from the joint effects of all formulas in the spreadsheet. Similarly, computation and communications in TeC are triggered asynchronously and the overall system behaviour is emergent.

A TeC system, called a *team*, consists of a collection of *players* with no central component responsible for coordinating each step of the action. Players are computing-enabled devices ranging from computers and smart phones, to smoke and motion detectors, to microwave ovens, clothes driers and smart electric meters (*www.smartmeters.com*). A team's overall function results from the joint effect of the *activities* carried out by the individual players.

Activities generalize the concepts of process, computational service, and function of a device. Specifically, activities may be of short duration with a discrete output, like service invocation; they may be long-lasting with a succession of inputs and outputs, like processes; or they may last for months without producing an output, like the function of a smoke detector.

End users may design different teams to serve purposes such as "surveillance," or "energy management." A *team design* includes *activity sheets*, describing the role of each player in the team, i.e. its activity, and *communication* paths for channeling asynchronous messages and data streaming between players.

Figure 2 illustrates a team design for managing energy consumption at the home of a hypothetical user, Bob. The team overview at the top includes four activities shown as labeled rectangles and their communication paths, which link *out*put events to *in*put events, respectively shown as 🟨 and ■. At the bottom, the figure details the activity sheets for smart meter and for manage power.

Users customize an activity sheet starting from an *activity type*, i.e. a generic description of features. Activity types define input events and internal *outcomes*, which may change value as a result of the device's operation. Specialized devices, such as a smart electric meter or a smoke detector would typically be shipped with activity types which might be defined by a vendor or result from a community standardization effort. For example, smart meter includes price per kilowatt in cents, while smoke detection might include Boolean outcomes such as smoke, carbon-monoxide, etc.

In the example, Bob defined two output events on smart meter: peak and low, which are respectively announced when the price exceeds 10 cent per kilowatt and when it comes back under that amount. Specifically, price>10 is the *triggering condition* for the peak output event. The syntax and operational semantics of TeC are formally specified in [17].

The activity sheet for manage power is derived from the generic type *activity*, which unlike specialized activity types, includes no definitions of input events or outcomes, and may be played by any computing-enabled device such as Bob's smart phone. In the example, Bob wants to set the thermostat to 78 F after a costly event is observed, and back to a cooler 74 F when energy is cheap. The desired temperature is attached as payload temp to the set event sent towards the thermostat.

Similarly to spreadsheets, the flow of values is best effort. Specifically, events may be freely interconnected and the matching of attributes in the payload is based on attribute name. While this saves users from the tedious matching associated with method invocation in mainstream programming languages, it may lead to unintended errors. To help users manage this tradeoff, players should have reasonable defaults to all attributes of input events. Furthermore, upon user request, the TeC editor (below) highlights which attributes are being matched in a connection.

To make it easy to design and deploy distributed teams, TeC combines automated discovery of players with explicit spatial constraints indicated in the form name @ space, to be read name *at* space. Spaces are identified in human-readable form as city/street-address/room. For example, fairfax.va-22030.us/456.windy-rd/kitchen.2 indicates the kitchen in apartment 2 at 456 Windy Road in Fairfax, Virginia. Furthermore, users may define aliases for a space, such as Bob's home (specific address not shown here), or a list, such as trusted spaces, enumerating the addresses where a user is comfortable deploying his or her teams.

In Figure 2, all the players within the shaded rectangle are to be found @ Bob's home, while manage power is unconstrained. That is, if Bob decides to deploy manage power on his smart phone, he can follow the events while away from home.

One implementation of a manage power player that includes a user interface (UI) is shown in [6]. Through that interface, users are informed of arriving input events

(*a*)

```
<activity-sheet type="smart meter">
   <out-evt name="peak" trigger="price > 10">
         <target in-evt="costly" ip="mp-ip" port="mp-p" key="mp-k"/>
   </out-evt>
   <out-evt name="low" trigger="price <= 10">
         <target in-evt="cheap" ip="mp-ip" port="mp-p" key="mp-k"/>
   </out-evt>
</activity-sheet>
```

(*b*)

```
<evt name="costly"/>
```

(*c*)

```
<evt name="set"/>
   <att name="temp" value="78"/>
</evt>
```

Fig. 3. Example messages for the team in Figure 2: (a) briefing issued by the TeC editor towards the player for the smart meter activity, (b) message costly issued by the latter after a peak event, and (c) message set issued by manage power after a setT event.

(e.g., costly or cheap energy) and may *complement* the automated behaviors expressed in the activity sheet with manual triggering of events to control the thermostat and dryer.

Furthermore, teams may include multiple players that carry out an identical activity: indicated in by a stacked box for the activity sheet. In Figure 2, multiple manage power activities may be deployed, for example enabling Bob's family members to help with energy management via their own smart phones. Semantically, events leading to a stacked activity sheet are multicast to all participating players, and events originating from a stacked sheet may be issued by any of those players.

Multi-user capabilities such as these benefit from additional mechanisms to help regulate conflicting users actions (section 5.3).

4.1 Tool Support

Teams are designed by users interacting with an editor. Two versions of TeC's editor are currently available: one for personal computers, built over the modeling framework of Eclipse (www.eclipse.org/modeling/gmp), and another for Android phones and tablets (www.android.com).

When a user decides to deploy or update a deployed team, TeC's editor triggers two operations: *discovery* and *briefing*. During discovery, concrete players for the activities in a team design are identified and confirmed to be up and running. TeC's middleware then briefs each of the players with the corresponding activity sheets: see example in Figure 3(a).

Once briefed, players interact with their physical environment and with each other and with no further intermediation or central coordination: see Figure 3(b,c). In the current implementation, communication takes place over TCP/IP, and is secured by symmetric encryption. A key is assigned to each player *p*, and shared with others that send messages towards *p* (specific values elided, shown in italics, in Figure 3).

More details concerning TeC middleware and systems can be found in [6], [18].

4.2 Adaptation Loop Closed by User

Users may edit team designs after deployment: Figure 1(c). If at any point Bob is not happy with the way the team in Figure 2 is working, he might rewire the events, change the activity sheets, add or remove activities. He may then ask the editor to redeploy the team. The editor compares the new design with the original and re-briefs players as needed.

This adaptation loop is made possible by TeC's automated deployment of design artifacts, and it is made easier by its declarative semantics which carries no computational state (see also the discussion on statefulness in section 7.2).

Although self-adaptive ubicomp systems will be illustrated throughout this paper using TeC, the self-adaptation principles are generalizable to other design languages that carry enough semantics to be automatically deployable.

5 Design Meshing

Design meshing is motivated by the combination of two growing trends: the ubiquity of computing, and the use of cyber-physical services.

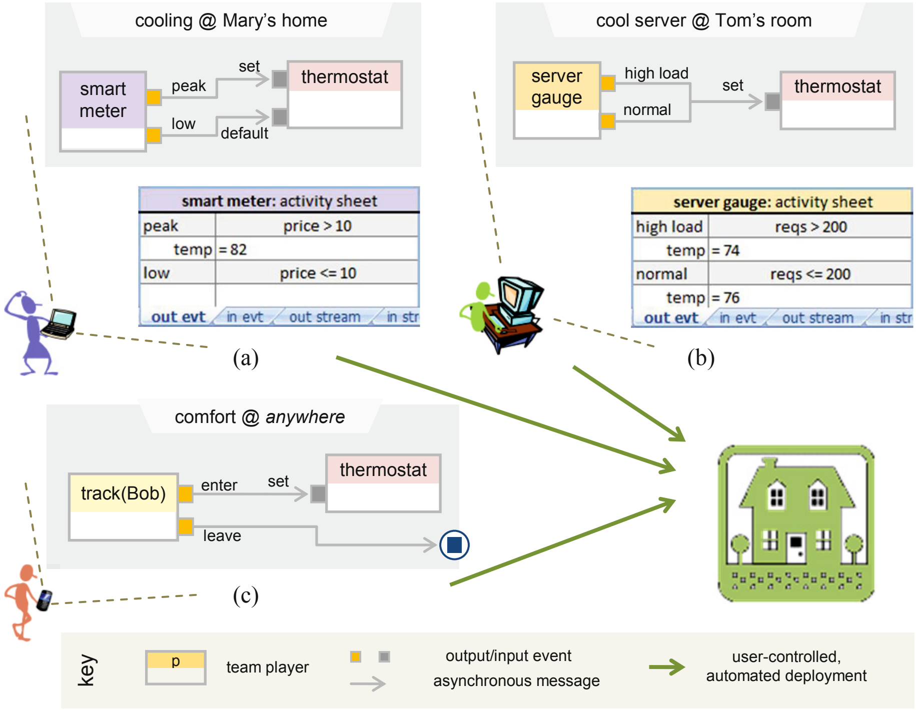
First, people access most physical spaces in far more spontaneous ways than traditional computer systems. While the latter require users to be administratively registered in advance, in ubicomp the preregistration of users with a space should be lightweight or even non-existent in cases such as customers entering a store.

From the users' point of view, increasingly the expectation is for spaces to make their services available to impromptu occupants, while respecting security and different levels of access. For example, a smart home may allow visiting friends to discover and use entertainment services, while it may allow a nurse to discover and interact with assisted-living sensors.

From a technical point of view, ubiquitous computing requires smart spaces to be able to dynamically change the features of the systems deployed therein to match user expectations, adding and dropping components and connections as needed, and interacting with the software deployed on mobile devices carried by users.

Second, cyber-physical services such as provided by thermostats, smart appliances, and other sensors and actuators are increasingly integrated in applications deployed in smart spaces. However, such *cp*-services are unlike purely software services with respect to sharing by multiple users. Software services may avoid side effects across users by creating a sandboxed replica of the service for each user. In contrast, setting a house thermostat to a given temperature has a lasting effect on the environment perceived by everyone in the house: the thermostat is effectively *shared* by all.

Design meshing combines the ability to *automatically* deploy and adapt systems in response to the comings and goings of users, with the ability to orderly share cp-services among users. In other words, the design of a space's ubicomp system at a given moment results from *meshing* design artifacts contributed by different users, and which incidentally share cp-services. To facilitate meshing, stakeholders define service-specific policies for resolving the sharing conflicts that may arise.



smart meter: activity sheet	
peak	price > 10
temp	= 82
low	price <= 10

out evt / in evt / out stream / in str

server gauge: activity sheet	
high load	reqs > 200
temp	= 74
normal	reqs <= 200
temp	= 76

out evt / in evt / out stream / in str

Fig. 4. Systems designed by three hypothetical users, Mary, Tom, and Bob, which will be automatically meshed when deployed due to sharing the services of the house thermostat

5.1 A Variety of Design Artifacts

Figure 4 shows an example with three separate design artifacts, all of which impact the air conditioning control at the house of a hypothetical user, Mary. These artifacts are deliberately kept simple to help focus on the meshing aspects.

The design in Figure 4(a) captures Mary's intention to automatically raise the thermostat setting to 82 F when peak electric rates are in effect. For that, Mary leverages pricing signals issued by her smart electric meter. Figure 4(b) shows a different artifact designed by Mary's son, Tom, who is finishing a computer games degree and is running his startup business from home. Tom installed a server that hosts a multiplayer game in his room and wishes to keep room temperature at 76 F, lowering it to 74 F when the server experiences loads above 200 interactions per minute. Figure 4(c) shows a third artifact designed by Mary's friend, Bob, who has developed a respiratory condition and does better in cooler, drier air. Bob would like the temperature in any space he enters to be set at 74 F.

The significance of this example is that each of the artifacts was independently developed by users, without knowledge of each other's designs or intentions. Similarly, Mary's smart home has no prior knowledge of which designs users will want deployed at a given time, or which designs will come into its space carried in mobile devices. For example, Mary may deploy her cooling design each spring when the weather warms up and then stop it in the fall; Tom may deploy the cool server artifact intermittently, taking it down during server maintenance; and Bob's design artifact only becomes known to the house once Bob comes to visit.

Design meshing automatically adapts the energy management system at Mary's home depending on which artifacts users want deployed at a given time, and reconciling the conflicts in thermostat setting requested by each artifact.

5.2 Understanding the Details

This section examines Figure 4 in the light of the notation and semantics in section 4, and in preparation for discussing the extensions towards meshing in section 5.3.

In Figure 4(a), Mary defined output events peak and low for the smart meter, with triggering conditions similar to the ones in Figure 2, but additionally defined a payload for peak and connected both to input events on the thermostat. Specifically, the payload of peak is used to set the temperature to 82 F, more relaxed and thus less energy demanding than the default of 78 F set by Mary directly on the thermostat.

The thermostat activity type defines an input event set which recognizes a payload labeled temp.

Figure 4(b) shows team cool server designed by Mary's son, Tom. Tom wrote a load gauge piece of software for the server and wrapped it as a TeC player, so that it could be discovered by the TeC middleware. Once the player and its activity type were recognized by the middleware, server gauge became available in the editor and Tom was able to include it in the design. Tom directed both output events high load and normal to the input event set in the thermostat, which interprets the payload temp of any incoming event.

Bob's comfort team in Figure 4(c) is meant to be deployed in any space Bob enters. To make that possible, TeC supports activities of type track, which take as a parameter the id of the entity to track, i.e. user or device, and may use a variety of concrete means to do so [19], [20]. Outcomes enter and leave are defined in the track activity type, and correspond to the tracked entity entering and leaving the space where the track activity is deployed. Bob used these outcomes as triggers for two corresponding events, and he added a payload temp = 74 to event enter. This activity sheet is not shown for brevity, but the result of this team design is that an enter event with temp = 74 is sent towards the space thermostat whenever Bob enters a space.

Teams may be associated with a space, or they may be mobile. The teams in Figure 4(a) and (b) are associated with fixed locations: Mary's home and Tom's room, which are indicated after the @ sign at the top of each team design.

TeC's middleware is capable of reasoning about spatial containment: for example, Tom's room is contained in Mary's home, and therefore, lacking its own thermostat, TeC will map the one in Tom's design to a suitable device in Mary's home.

To make his team mobile, Bob keeps the design in Figure 4(c) on his smart phone. He used the keyword anywhere as the team's location constraint, but he could have used a user-defined alias such as trusted spaces (section 4). Once Bob tells the TeC editor on his phone to deploy team comfort, each time the phone recognizes a new

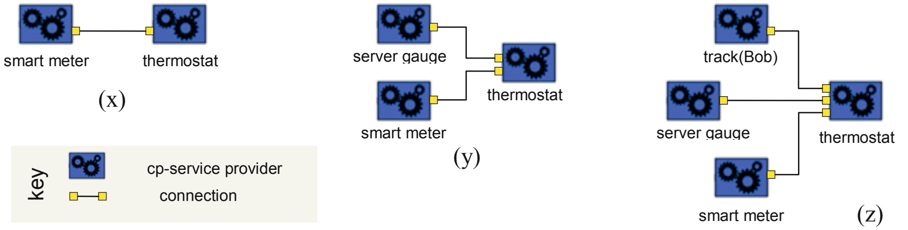


Fig. 5. System configurations after meshing the designs in Figure 4, respectively, (a) only; (a) and (b); and (a,b) and (c)

space, the editor checks the location constraint and upon a match it tries to find appropriate players and deploy the team (more about space-awareness in [18]).

To automatically stop, i.e. "undeploy," the team at a space when he leaves, Bob connected the leave event issued by track(Bob) with the stop operator ⏹ for the team (more about this in section 6). Until Bob tells the TeC editor to no longer deploy comfort, it will keep deploying a new instance of the team at each space Bob enters.

5.3 Meshing

Figure 5 illustrates three configurations at Mary's home that result from different combinations of the team designs in Figure 4. The result in this example is not a collection of isolated systems, one for each user, but *one* system coupled by the incidental sharing of the thermostat. This system *adapts* in response to the deployment/withdrawal of design artifacts.

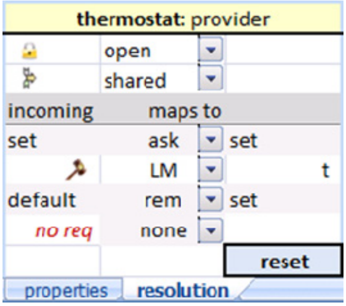
The incidental sharing of providers for cp-services, players in TeC, may raise *conflicts* in the desired services settings, which must be resolved in ways that users can understand and perceive as fair.

In general, conflicts arise because operations on cp-services are expected to have a lasting effect on the provider and its surrounding physical environment: specifically, for the duration of the cp-service, or until the same team invokes an overriding operation. For example, while the team in Figure 4(b) is deployed, Tom would expect the thermostat to be set to 76 F during normal loads and to 74 F during high loads. However, as per Figure 4(a), Mary expects the thermostat to normally be set to 78 F and as high as 82 F during peak energy prices.

Design meshing includes mechanisms to resolve conflicts characterized by a desired value, such as the channel on a radio or TV, the sound volume on the same, the level of lighting in a room, or the temperature set on a thermostat, as in the example above.

Specifically, the stakeholders of a space may establish *policies* for each of the service providers therein, which may be customized during provider deployment and updated as needed. These policies cover three aspects:

- *Access* 🔒 : may be open to any visiting user on a pseudonym basis, or may be restricted to a set of pre-registered authorized users. In the latter case, the requesting team will facilitate providing the necessary credentials to the cp-service provider.



thermostat: provider		
🔒	open ▾	
	shared ▾	
incoming	maps to	
set	ask ▾	set
	LM ▾	t
default	rem ▾	set
no req	none ▾	
		reset

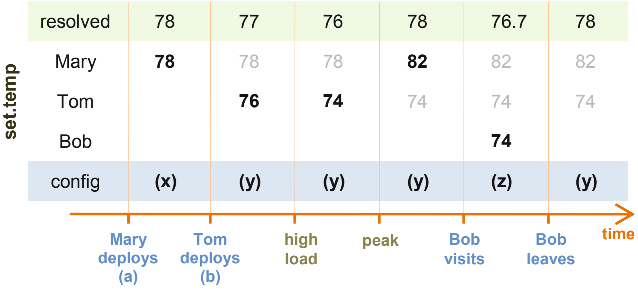
Fig. 6. Resolution sheet for the thermostat at Mary's home

- *Multiplexing* : shared providers may participate in several systems simultaneously, while exclusive providers participate in only one team at a time.
- *Resolution* : different rules are available to resolve requests on shared providers, which keep a list of outstanding requests. Priority will preempt requests from lower priority users in favor of the one with the highest priority, while majority takes the statistical mode of the outstanding request's parameters. Both frequent and infrequent keep tallies of user requests, freq giving precedence to requests from frequent users, similarly to a frequent customer program, and infreq giving precedence to infrequent users, similarly to the notion of fairness in round-robin mechanisms. Least Misery is applicable to requests pertaining quantities that result in the least discomfort for all requesters when averaged.

Figure 6 shows the default policy associated with the thermostat activity type. Access is open and shared, allowing every user with access to a space to express their preferences with respect to cooling. Resolution of the set payload is set to the LM policy, so that conflicting requests are resolved by averaging.

Although not required to, stakeholders might customize this policy to suit their needs; for example changing the access to authorized and listing the authorized users, or changing the resolution rule to priority and listing the priorities of users. Changes may be made at any time by an authorized stakeholder, who might also press the reset button to retrieve the type's default policy.

Figure 7 shows an example of the adaptations and conflict resolution for a simple scenario. Time runs horizontally from left to right: initially Mary deploys the team in



	resolved	78	77	76	78	76.7	78
set.temp	Mary	78	78	78	82	82	82
	Tom		76	74	74	74	74
	Bob					74	
	config	(x)	(y)	(y)	(y)	(z)	(y)

Fig. 7. Timeline of adaptation with resolution, for configurations (x,y,z) in Figure 5

Figure 4(a), at some point Tom deploys the team in Figure 4(b), later a high load event is issued, etc. The configuration row indicates the system configuration, as shown in Figure 5, that results from each event. The rows labeled set.temp show the payload of events set issued by each team, with new incoming requests highlighted. The row at the top shows the value for set.temp, after applying the resolution policy in Figure 6. Users will notice cooler air when Bob is present and when Tom's server has high loads, although their original requests are automatically weighted against Mary's concern with peak energy prices.

5.4 Under the Hood

A provider's resolution mechanism keeps a list of outstanding requests per each input event in the activity. In general, requests issued by a team override previous requests by the same team. Also, requests issued by a team are automatically dropped when the team is withdrawn: the middleware sends a signal to the providers to discontinue all of the team's activities.

Resolution sheets define a mapping from incoming events to actual events processed by the provider. Frequently, these rules are straightforward, i.e. the mapping does not change the event. However, the interplay between set and default in the thermostat makes the mapping less trivial. Specifically, if a team sends a default event, that is mapped to the removal of its previous set request, which has a similar effect whether or not the team is the sole client of the shared provider. Figure 6 shows the mapping of set onto itself, and the mapping of default to either (a) removing the previous set issued by the team, or (b) the suppression of the event, if no previous request was made by the team.

5.5 Scalability

Frequently, adaptation mechanisms rely on global optimization algorithms which have a complexity that grows, often non-linearly, with the size of the system.

In contrast, meshing decisions are made locally by each service provider, i.e. a player in TeC, and the overall system structure emerges from such decisions.

Specifically, when a user decides to deploy a design artifact at a space, the editor discovers and briefs each player. Users and their supporting middleware need not become aware of which other users and design artifacts are competing for services at the space, for privacy reasons, but may become aware of the resolution policies of the discovered players. It is upon each player to respond to the discovery enquires of different editors, and if its multiplexing policy allows, to mesh the briefings.

Furthermore, meshing may result from a single design artifact such as the one in Figure 2. In that example, several manage power activities may be deployed, e.g. on the smart phones of Bob's family. Because of their links to a stacked box, the briefings sent to the players of thermostat and dry inform them that conflicting requests may be received from different manage power players. To resolve those, the stakeholders at Bob's home might have agreed on LM and priority policies respectively for the thermostat and clothes drier (see Figure 6).

That is, meshing policy is defined for each player deployed at a space. Players come with default policies for their activity type, which can be configured by the stakeholders at the space, as desired.

In a nutshell, meshing scalability is contingent neither on the complexity of the design artifacts to be deployed, nor on the size of the system that results of the meshing of such artifacts.

6 Pliable Apps

Propelled by mobile computing, adaptation to context, aka *context-awareness*, has been the prevalent form of self-adaptation in ubicomp [21]. Context-aware applications reconfigure their behaviors depending on sensed context such as user presence at a location, physical activity (walking, driving...), social activity (in a meeting...) etc.

However, adaptation policies were often decided before deployment and fixed in the code, becoming hard to change after the system is deployed. The usability community has observed that this causes users to perceive adaptive systems as *rigid* and to feel at the mercy rather than in control of technology [22].

Design constructs such as illustrated in section 4 are an important step towards enabling users to tailor adaptation: users may design parametric changes to the behavior of systems, e.g. the clothes drier adapting to energy prices. However some adaptation requirements may require deeper, structural changes to an application.

Pliable apps allow users to, first, design *structural adaptation*, and second, to change the way it works after deployment – Figure 1(a-c). For that, we extend machine-interpretable design notations with (a) constructs to describe adaptation triggering events, such as context events, and (b) constructs to explicitly manipulate the run-time structure of an application.

6.1 Designing Structural Adaptation

Suppose that Susan, Mary's elderly mother, developed a heart condition. Her doctor allowed Susan to return home as long as her heart is under constant monitoring by an assisted living system that automatically calls Mary, and failing that, 911, should a situation of concern arise. Furthermore, because Susan may become disabled in such circumstances, video should be streamed along with the phone call to help responders evaluate and respond more effectively to the situation.

Susan discussed the tradeoff between safety and privacy with Mary and they agreed that the team should only place a phone call, and especially stream the video, if Susan is alone. From a technical point of view, this system is structurally self-adaptive: it only exhibits the feature that makes calls in response to a situation of concern when Susan is alone.

Figure 8 shows a TeC system, a team (see section 4), that embodies these assisted living features. The monitor heart activity will issue concern events, which always sound an audible alarm, and are additionally directed to activity make calls.

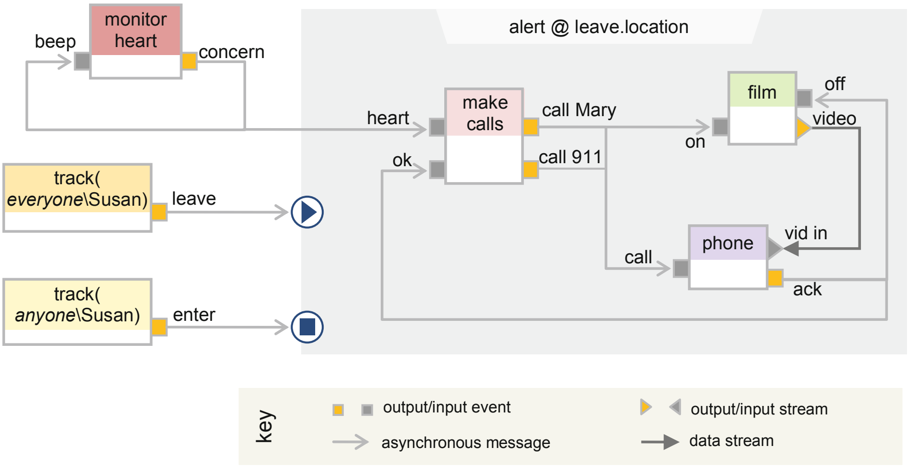


Fig. 8. Monitoring Susan's heart condition

Mary defined as a subteam, alert, the part of the design concerning the calls and video: the shaded rectangle in Figure 8. Make calls issues two events directed at phone, which carry as payload the phone number to dial and a short informative message for a human on the other end of the line. Of these events, call Mary is triggered as soon as heart is observed, while call 911, reacts to heart by waiting a short while for reassurance, and in the lack of it contacts the city's emergency services. To understand the details, the triggering condition for call 911 uses operator P, which postpones the observation in the first parameter by the period specified in the second parameter, and operator T, for toggle, which becomes alternately true or false once the first or second conditions are observed, respectively [17]. In the example, P(heart,0:01) sets a one minute time out after heart is observed, and T(heart,ok) disables it once ok is observed.

Mary may reassure make calls, and avoid a call to 911 during a false alarm, by pressing key '5' when she receives the phone call. Recognizing key tone '5' on the phone line is used as a triggering condition for event ack (the activity sheet for phone is not shown, for brevity). Either of the call events will also start video streaming (event on in film), which is stopped once an ack event is issued.

Structural adaptation works as follows: subteam alert is deployed when everyone but Susan leaves the space where Susan is, and it is retired when anyone but Susan enters that same space. To understand the details, generic keywords everyone and anyone are defined in the track activity type, and Mary decided to use them. Instead she could have listed trusted people, whose presence should withdraw subteam alert.

TeC defines constructs start ▶, to deploy a (sub)team, and stop ■, to retire a (sub)team. In the example, whenever subteam alert is not deployed, the players for phone, make calls, and film keep working and possibly participating in other teams: they are just not part of Susan's assisted living team.

Figure 9 illustrates an adaptation scenario, where time moves from left to right. Mary is in the room with Susan when the assisted living team is first deployed,

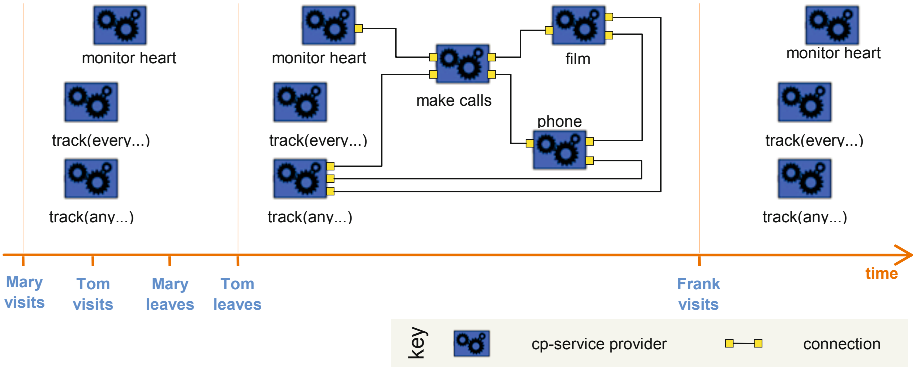


Fig. 9. Timeline of structural adaptations for Susan's team in Figure 8

therefore only the sensing part of the team in deployed. Subteam alert is automatically deployed when Susan's last visitor leaves, and it automatically withdraws when Susan's friend, Frank, comes by. Should a concern event be triggered, it will cause Susan's heart monitor to beep, but Frank will help decide whether Susan's heart rate is due to healthy excitement, or if there is cause for alarm.

6.2 User Control of Adaptation

The constructs above are at a much higher level of granularity than programming constructs for controlling the run-time creation/destruction of data structures (malloc, new...) and of processes (spawn, fork-exec...). They are also at a coarser level than the operators to add/remove individual components and connectors, commonly used to manipulate run-time architectural descriptions of adaptive systems, e.g. [11]. The underlying reason is to make it *easier* for users to express feature-wide adaptations as a unit, and to bring out a view of the different configurations under different circumstances while highlighting the events that trigger those adaptations.

For example, Mary and Susan might decide after some time that only video streaming should be excluded from the system when Susan has company, but calls should be made every time there is cause for concern. For that, Mary could redefine the scope of adaptation in Figure 8 by reshaping the boundaries of subteam alert to include only activity film. No other changes would be necessary.

Although Figure 8 uses TeC syntax, it illustrates more general concepts for user-controlled structural self-adaptation. Namely, machine-interpretable design notations may describe adaptation at the grain of subsystems which is triggered by application-level events. Such events may originate in context sensing, e.g. associated to tracking the presence of users, or in gauging system metrics such as performance or load, e.g. the server gauge in Figure 4(b).

What makes these kinds of adaptations *user-controlled*, is the fact that adaptation is described at the design level using machine-interpretable constructs. Changes made by users to the design of adaptation can be automatically translated and (re)deployed at run time.

7 *B*self-Healing

The kinds of systems that we have discussed so far, and that characterize ubicomp, are deployed over a set of collaborating devices, such as cell phones, small computers, smart appliances, cameras, smoke detectors, etc.

This kind of system organization is in contrast with both the *client-server* and with the *cloud computing* models. Those assume the availability of powerful and reliable server infrastructures, even if the identity of such servers is dynamically bound and therefore "transparent" to clients, as in cloud computing. In contrast, ubicomp systems may include an occasional powerful device, but no assumptions are made about the constant availability of server infrastructures.

As discussed in section 2, ubicomp systems are often *open* and deployed in *harsh* environments: in open systems, new components/devices may join the system, and in harsh environments *any* system component may fail or become disconnected.

These characteristics cause mainstream solutions for self-healing to become brittle and/or hard to scale in ubicomp. Classical approaches such as Rainbow [11] factor out self-healing and self-optimization capabilities to a small set of dedicated components. These components are normally deployed with the servers they monitor, or they are deployed in one node and given oversight over the distributed system to be monitored. Unfortunately, such *centralized* solutions present a single point of failure.

To avoid that, some solutions *replicate* the knowledge about the whole system and healing capabilities on some or all nodes [13]. Unfortunately, replication creates complexity: coordination protocols must guarantee that all nodes have a consistent view of the system, reach consistent decisions, and don't interfere with each other when performing recovery actions. This approach does not fit ubicomp well, given the limited resources on most nodes.

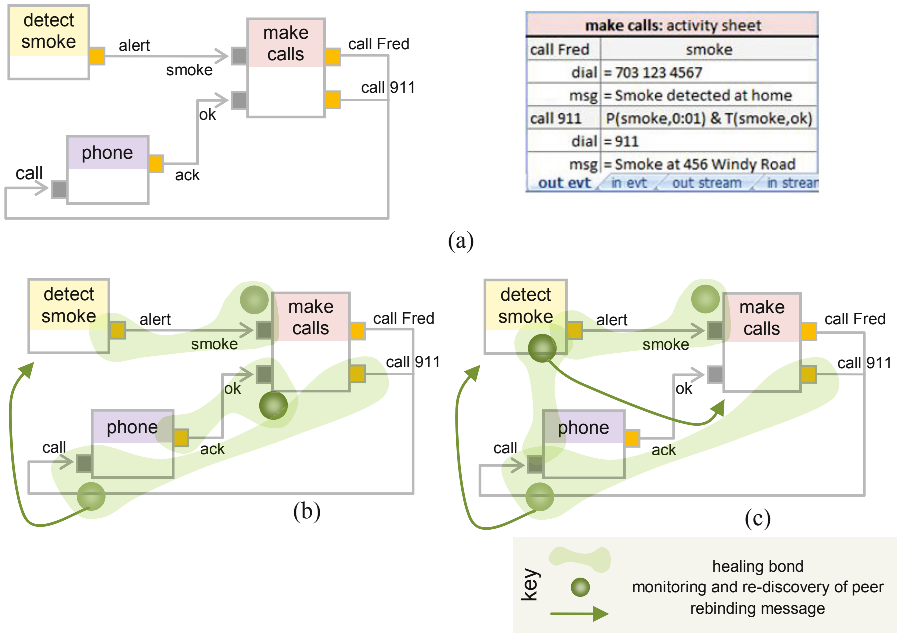
Created with ubicomp in mind, PCOM proposes a *client-centered* approach: the middleware interprets the *requires* spec of a node and automatically finds a replacement when a faulty service is invoked [23]. Unfortunately, a synchronous call-return style of interaction is only part of the story: some ubicomp nodes are meant to work autonomously and send messages asynchronously, e.g. a smoke detector, other nodes are meant to stream data, e.g. a camera, etc.

*B*self-healing recasts failure recovery for devices with limited processing, energy, and communications, and which may interact in a variety of styles. Also, since users may arbitrarily turn off devices, forget to change batteries, or disconnect cables while cleaning the house, *B*self-healing is resilient to failures and disconnections of any components, including those with self-healing responsibilities.

To illustrate *B*self-healing, Figure 10(a) shows a TeC system (section 4) for reporting smoke at the house of a hypothetical user, Fred. A smoke detector may issue alert events which are picked up by input smoke in make calls. The latter and activity phone work identically to the ones in Figure 8.

7.1 Healing Protocol

In *bond* self-healing, *B*self-healing for short, each node *p* is assigned a *healer*: another node who (a) receives enough information to recover *p*'s activity should *p* fail; and (b) is responsible for monitoring *p*. The two nodes thus have a healing *bond*.



make calls: activity sheet	
call Fred	smoke
dial	= 703 123 4567
msg	= Smoke detected at home
call 911	P(smoke,0:01) & T(smoke,ok)
dial	= 911
msg	= Smoke at 456 Windy Road
out evt	in evt / out stream / in strear

Fig. 10. Example smoke detection system (a) with two possible healing bond configurations, (b) and (c)

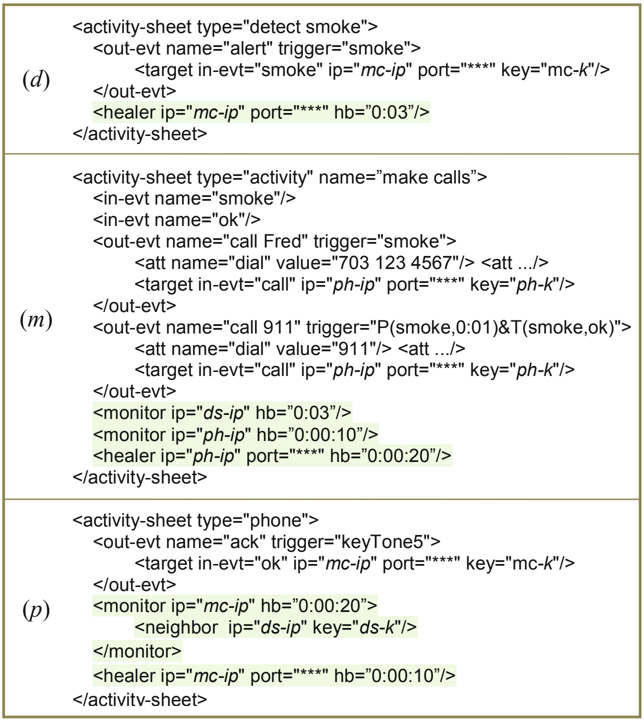
Figure 10(b) shows a possible set of healing bonds for the team in Figure 10(a). Bonds are depicted informally as bone-resembling blobs with a darker circle, the "marrow," on the healer side. This set of bonds assign make calls as the healer for detect smoke and for phone, and phone as the healer for make calls.

Figure 11 shows the player briefings for the activities in Figure 10. In addition to encoding the activity sheets in Figure 10(a), the highlighted parts capture the healing bonds in Figure 10(b). For example, briefing (*p*) informs the player for phone of two healing bonds: first, it should monitor a player at an IP address represented in the figure as *mc-ip*, and second, its own healer can be found at address *mc-ip*, incidentally the same address, given the configuration of bonds.

A healing bond establishes a heartbeat protocol between a player and its healer: in the example, phone expects to see a heartbeat message originating from *mc-ip* every 20s, and conversely, it should send a heartbeat message towards its healer every 10s. In case of failure, a healer is also responsible to send a rebind message to the neighbors that send messages towards the monitored player: in the example, the neighbor element in (*p*) informs phone of that responsibility.

Figure 12 illustrates the *B*self-healing protocol with a scenario. Fred used his smart phone to edit and deploy the team in Figure 10(a). The editor discovered a smoke detector in Fred's kitchen and a landline phone nearby. Activity make calls was assigned to a generic *activity* player also running on the smart phone.

Therefore, TeC's editor sent briefing (*d*) to the smoke detector: Figure 12 left-hand side. It also sent the landline phone both (*p*) for its own activity, and (*m*) for its role



(*d*)

```
<activity-sheet type="detect smoke">
   <out-evt name="alert" trigger="smoke">
         <target in-evt="smoke" ip="mc-ip" port="***" key="mc-k"/>
   </out-evt>
   <healer ip="mc-ip" port="***" hb=”0:03”/>
</activity-sheet>
```

(*m*)

```
<activity-sheet type="activity" name=”make calls”>
   <in-evt name="smoke"/>
   <in-evt name="ok"/>
   <out-evt name="call Fred" trigger="smoke">
         <att name="dial" value="703 123 4567"/> <att .../>
         <target in-evt="call" ip="ph-ip" port="***" key="ph-k"/>
   </out-evt>
   <out-evt name="call 911" trigger="P(smoke,0:01)&T(smoke,ok)">
         <att name="dial" value="911"/> <att .../>
         <target in-evt="call" ip="ph-ip" port="***" key="ph-k"/>
   </out-evt>
   <monitor ip="ds-ip" hb=”0:03”/>
   <monitor ip="ph-ip" hb=”0:00:10”/>
   <healer ip="ph-ip" port="***" hb=”0:00:20”/>
</activity-sheet>
```

(*p*)

```
<activity-sheet type="phone">
   <out-evt name="ack" trigger="keyTone5">
         <target in-evt="ok" ip="mc-ip" port="***" key="mc-k"/>
   </out-evt>
   <monitor ip="mc-ip" hb=”0:00:20”>
         <neighbor  ip="ds-ip" key="ds-k"/>
   </monitor>
   <healer ip="mc-ip" port="***" hb=”0:00:10”/>
</activitv-sheet>
```

Fig. 11. Briefings for the activities in Figure 10(a), including the healing bonds in Figure 10(b) – highlighted

as the healer of make calls. Likewise, the editor sent the generic player on the cell phone both (*m*) for its activity, and (*d*) for healing detect smoke.

At some point, Fred notes that his cell phone is running low on battery and turns it off, planning to look for the charger later: Figure 12, center. The landline phone notes the absence of make calls' heartbeats, searches for a player for a generic *activity* and finds a smart microwave. It then sends briefing (*m*) to the microwave and a rebind message to all of make call's neighbors, namely the smoke detector at *ds-ip*.

Upon receiving the rebind message with the IP address of the new player for make calls, the smoke detector realizes it has a new healer. Accordingly, it sends its own briefing (*d*) to the microwave, so that the latter could recover detect smoke, should the need arise, for example by discovering a smoke detector in another part of the house. Likewise, the landline phone attaches a phone recovery briefing (*p*) to the message sent to its new healer.

Later, the landline phone cable gets accidentally disconnected while cleaning the house: Figure 12 right-hand side. The microwave, phone's healer, notices the failure and looks for a new player for the phone activity type. It finds a voice-over-IP application running on Fred's home office computer, and proceeds to send it a phone briefing (*p*) plus a recovery briefing (*m*), as part of its role of healer for make calls.

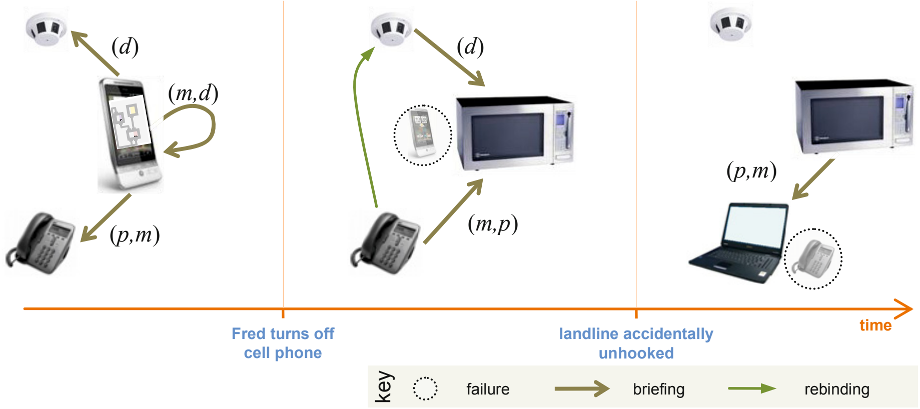


Fig. 12. Timeline of *B*self-healing for the team in Figure 10(b), facilitated by briefings (*d*,*m*,*p*) as defined in Figure 11

7.2 Architectural Properties

Resilience. We argued above that decentralized solutions for self-adaptation avoid a single point of failure, and therefore are a better fit for the harsh environments in ubicomp. Because self-adaptation requires run-time service discovery in open domains, we must examine the architectural properties of discovery mechanisms.

There exist two mainstream styles of discovery: directory-based and broadcast-based [3]. In the first, components register their services with a directory, which then brokers incoming requests by type while possibly factoring in optimality criteria expressed by the requester, e.g. with respect to quality of service or physical proximity between requester and provider [24]. Unfortunately, this creates a critical dependency on the directory hosting device and its network connectivity.

In broadcast-based discovery, service requesters broadcast their needs and then select a provider among those who answer. However, to avoid flooding the internet, a historic design decision at the level of the internet protocols limit broadcast to the boundaries of the local network where it is issued; that is, network routers and bridges drop broadcast messages.

*B*self-healing employs broadcast-based discovery, thus avoiding the creation of a single point of failure on the service directory. To scale discovery past network domain boundaries we are working on design notations to scope discovery to concrete locations [18], which will enable resolvers on the network infrastructure to tunnel the broadcast to the appropriate local networks. This is the topic of ongoing work.

Recoverability. There is a tradeoff between recoverability in distributed protocols and protocol footprint. On the high end of recoverability, each component holds a replica of knowledge about the whole system and can recover any other component. However, the protocols must guarantee that all healers have a consistent view of the system, reach consistent decisions, and perform consistent recovery actions.

Since each healing bond pertains to the recovery of only one component, the *B*self-healing protocol is much simpler than the solutions above. It is therefore a better fit for small devices.

The downside occurs when a component and its healer both fail. In the example in Figure 12, if detect smoke fails while make calls is being recovered, then the knowledge of (*d*) is lost and detect smoke will not be recovered. Nevertheless, this protocol can recover from any isolated failure and many cases of compound failures.

Of course, this is a statement about the protocol, not about the environment in which the system is running: if the environment lacks alternative providers for the failed features, then no protocol, no matter how sophisticated, will be able to heal the system.

Models footprint. The knowledge held by a component follows the tradeoff above, it decreases with simplicity but it is never zero. Even without self-healing, a component knows who are its neighbors and the rules governing communication, e.g. contracts, or in the case of TeC, activity sheets.

In *B*self-Healing, a component is additionally aware of, first, its healing bonds: the monitor and healer elements on the sheets in Figure 11. Here, the placement of the healing bonds along the application's connectors enables piggybacking of monitoring information and reduces the distribution of encryption keys. Second, a healer is also given the necessary knowledge to perform healing: in TeC, the monitored component's briefing, e.g. (*d*) being passed to the microwave in Figure 12.

As Figure 10(b) and (c) suggest, there is more than one way to assign healing bonds. The assignment algorithm that produced the bonds in Figure 10 (b) follows the communication paths already in the design, which enables piggybacking heartbeats on application messages, an advantage in case of frequent messages, and reduces the distribution of encryption keys.

The algorithm that produced (c) avoids mutual bonds, as between phone and make calls in (b), thus improving recovery from double failures. For example, if phone and make calls both fail, in (b) neither will be recovered, while in (c) detect smoke will still be able to recover phone. The loss of a single component is still crippling in this example, but a larger system might be able to work in a degraded mode without one of its components.

Irrespectively of the placement of bonds, when a component *p* is being recovered, the knowledge to necessary for *p* to perform healing is being passed by the components to be monitored by *p*, rather than by *p*'s healer. This avoids a recursion problem: in the example, if phone were to have the complete knowledge to heal make calls, that would include the knowledge for make calls to heal detect smoke. In a larger system, this recursion might produce long chains which would jeopardize both protocol footprint and being resilient to partial losses.

Statefulness. Providers are stateless in many service-oriented systems, including TeC's, thus simplifying the protocols and knowledge necessary for healing.

The proposed solution may be extended by having checkpoints of a component's state sent periodically to its healer, e.g. piggybacked on the heartbeat messages. A similar idea supports the follow-me mechanism in Project Aura, which suspends a service in one provider and resumes it in another by transferring a markup representation of the service's state [14].

Responsiveness. There is also a tradeoff between the responsiveness of healing and the ability to use mobile devices: the higher the heartbeat rate, the quicker the detection and healing, but the higher the communication overhead and the draw on batteries. In hostile environments, a high heartbeat rate also increases the chances of adversarial localization of a mobile device, as well as the chances of cracking encryption by brute-force attacks.

*B*self-healing has each component propose a maximum supported rate during discovery, which may be confirmed or relaxed during briefing (monitor elements on the sheets in Figure 12). At any rate, recurring communication losses may lead a healer to believe that a monitored component has failed, and to inadvertently create a zombie by rebinding its neighbors to a replacement. The neighbors may detect that an incoming message comes from a zombie, by virtue of keeping track of previous network identifiers, and reply to it saying that it is no longer part of the system. However, in domains such as search and rescue, messages from zombies may carry important information thus calling for further work on the treatment of zombies.

8 A Style for Adaptation in Ubicomp

The kinds of self-adaptation discussed in this paper and the strategies for their implementation build on a few principles shared by many approaches for self-adaptation, including TeC. Except for these principles, the particulars of TeC syntax and semantics are incidental to the discussion, and TeC terminology is used merely for convenience. Specifically, the principles are:

- *Service-orientation*: systems are built out of generic parts which are pre-deployed and running somewhere on the network.
- *Service discovery* carried out by automated mechanisms at run time, whenever there is call to find a new provider for a desired service/feature.
- *Automated deployment* of systems based on models that are precise enough to be interpreted by automated tools. This builds on automated service discovery and includes capabilities to fit a generic part to a system-specific role. Examples of such fitting include briefing in TeC, activation in Aura [14], and service invocation/interconnection, e.g. [5], [11].

The remainder of this section discusses strategies for implementing the kinds of self-adaptation in the previous three sections on top of these principles.

8.1 Impact of Decentralization

We have argued that the pursuit of resilience leads away from centralized components and pushes responsibilities towards system components. Part of this argument was that resilient solutions based on replication/redundancy of system-wide features, while appropriate for servers and cloud computing, are awkward for ubicomp.

Decentralization does not mean that self-adaptation is being pushed out of middleware: it may reside with the pieces of middleware distributed with system parts, and which facilitate their integration into the system. Decentralization does mean that each part gets local/neighborhood responsibilities, as opposed to the system-wide responsibilities in replication-based solutions.

This has a significant impact on all mechanisms related to self-adaptation. A large body of work, including all of the author's prior work [14], [17], [18], [25], relies on centralized components. The impact on *discovery* and on *self-healing* was already discussed in section 7, but it also impacts structural adaptations for *pliable apps*.

With a centralized approach, structural adaptations could be facilitated by a central *deployer*. The deployer would hold the parts of a design to be dynamically deployed and it would register to receive their start ▶ and stop ⏹ events. For example, in Figure 8, the deployer would hold the briefings for all the alert subteam, and the briefings for both track activities would ask to direct the enter and leave events towards the deployer. Upon reception of an leave event, the deployer would discover players for, and deploy all the activities in subteam alert. Similarly, upon enter the deployer would instruct all those players to stop their activity for alert.

In a decentralized approach, that role may be picked up by (the middleware on) the players themselves. Back to Figure 8, the track player that issues an event towards start ▶ holds all the briefings for all the alert subteam. When leave is triggered, track discovers players for each of the activities and sends them the corresponding briefings. It will also inform the player which issues an event towards stop ⏹ of the identities of those players so that the latter can stop them when the time comes.

This does not represent a significant increase of the capabilities required from individual players, since *B*self-healing already requires them to discover other players; specifically, replacements for the monitored components. Also, it does not create a critical point of failure on the player that starts a subteam, since its briefing now includes the briefings of the started subteam, and is exchanged with and recovered by its healer, as illustrated in Figure 12.

8.2 Diversity in Triggers and Changes

A key aspect of self-adaptation is capturing the changes to be made to the system and what triggers those changes. In Rainbow, for example, changes are described as adaptation scripts written by domain experts, and triggers are either user-defined QoS goals, for B3 self-adaptation, or failures for B2 [11].

The different kinds of self-adaptation in this paper capture triggers and changes distinctly. This strategy seeks the most effective solution for each case, and is in contrast with seeking a one-size-fits-all solution for all kinds of adaptation.

In design meshing, changes are described as design artifacts, such as in Figures 2, 4, 8, and 10, and triggers are user instructions to deploy or withdraw system support for those artifacts. In conflict resolution, changes are derived from the resolution policy associated with each shared component, Figure 6, and therefore are described independently of specific design artifacts, i.e. features, that users want to deploy. Triggers are application-level requests made by each user's (part of the) system.

In *B*self-healing there is no explicit description of changes, since they always concern the replacement of a failed component by another for the same service type. The only description is one that affects the responsiveness of adaptation and concerns the rate of hart-beat supported by each component.

In dynApps, changes are explicitly described in the application logic, i.e. in the design artifacts, since the point is to turn on or off features of the application in response to application-level events. In other words, the application logic cannot be understood without grasping the changes that take place and what causes them.

9 Related Work

Context-awareness is the prevalent form of self-adaptation in ubicomp [21]. Examples include smart phone-based reminders and navigation [26]; medicine cabinets [27]; and plant care solutions that examine sensor data, consult species-specific care instructions in a plant encyclopedia service, and post watering tasks to robotic gardeners [28]. Larger applications have been built to support the work of medical and nursing staff in hospitals [29], [30].

Here, researchers have focused on building prototypes pushing the limits of technology and evaluating user acceptance and engagement. From a software engineering point of view, these are custom-made applications where the components, interconnections, and permissible adaptations are carefully crafted and controlled at code level.

The agents community has worked on adaptation to diverse execution environments by providing agents that have different algorithms but similar responsibilities [31]. In other words, this body of work focuses on making sure that adaptation is possible by *enriching* the execution environment with redundant components for the same feature.

The self-adaptation community focuses on the principled design of *mechanisms* to detect need for, and to carry out adaptation, under the assumption that the execution environment has redundant capabilities readily available [32].

The very few examples of applying such mechanisms to ubicomp include PCOM [23], already mentioned in section 7, and the application of ArchJava to the robotic plant care mentioned above [33]. These examples propose *client-centric* approaches to self-healing, i.e. a 'main' component replaces invoked services when they fail. Although a step in the right direction, these solution make no provision to protect the 'main' component. Also, the nature of components and interactions in ubicomp is more diverse than service invocation, to include asynchronous messages and media streaming, which are not covered in these examples.

*B*self-healing makes sure that every component is monitored by a peer, regardless of the style of interaction or even regardless of the existence of a functional interaction between the two. *B*self-healing shares with these examples the strategy that each component has local/neighborhood responsibilities, as opposed to the system-wide responsibilities in replication-based solutions [13] and in centralized solutions [11].

Table 1 made the case that self-healing (B2) is only a part of a wider landscape of self-adaptation. Pliable apps (B4) are distinct from self-modifying code (C4). A self-modifying program may alter some of its own code, either as a result of initialization parameters or upon reaching a certain state, usually to improve performance [34]. In contrast, the design of pliable apps is not self-modifiable, but it contain rules for modifying the run-time organization of the system.

Pliable apps share column B with classical work in self-healing and self-optimization [2], [11], [12], [35], that is, they effect changes to the run-time structure. However, pliable apps differ from the classics both in the stimuli for adaptation and on the way that changes are described. In the classics, adaptation is triggered by system-level reasons such a failures and degraded performance. The changes are described as scripts, plans, or algorithms designed by engineers and tucked away in components dedicated to self-adaptation.

In pliable apps, adaptation is triggered by domain events and the changes correspond to modes of operation in the application logic. Therefore the description of stimuli and the changes they trigger come front and center to system design. Designing variations of *behavior* with system state is fundamental to programming and to state machines, but in pliable apps the changes are *structural* changes to the run-time organization of a distributed system. Furthermore, *pliability* also applies to adaptation itself by allowing users to tailor it to their needs, and to change adaptation at run time: a control loop explicitly closed by users, as in Figure 1(c).

Design *meshing* performs B7 self-adaptation since the stimuli are design artifacts, both the arrival of new artifacts and the evolution of deployed designs. This kind of adaptation complements classical B2-3 self-adaptation: while B2-3 is meant to *maintain* system features and QoS in the face of adversities such as failures, resource shortages, and increased workloads [2], [11], B7 is meant to *change* system features in response to the desires of (multiple) users.

Design meshing is different from Aspect-Oriented Programming (AOP) and from Software Product Lines (SPLs). AOP starts with a set of modules that encode different concerns and weaves them at join points to create one program with a coordinated set of features [36]. In design meshing, the set of features pertaining to each design artifact is separate and may be deployed and withdrawn independently from others. SPLs start from a parametric model with variation points and instantiate it multiple times to generate a family of separate systems [37]. Design meshing starts with a number of unrelated design artifacts produced by independent users and combines them taking into account the incidental sharing of cp-services, for which separate replicas cannot be created for each user.

This sharing leads to conflicts, which are handled by A7 self-adaptation: *resolution* changes parametric behaviors of the system based on the various and varying intentions expressed in design artifacts. Other communities have addressed conflict resolution: in economics, *auctions* support a form of negotiation among multiple parties competing for a limited resource [38]; the agents community developed protocols for multi-attribute *negotiation* [39]; and *mechanism design* is a subfield of game theory that investigates incentive-punishment mechanisms for promoting desired behaviors among independent actors [15]. The resolution mechanisms described in section 5.3 draw on that work for policies that can be *applied* automatically at run time: a feed forward control loop. By promoting the *choice* of policies to the design level, and which can be changed at run time, this mechanism supports an additional control loop, now explicitly closed by users.

10 Conclusion and Future Work

Ubiquitous computing is an exciting area to work on self-adaptation. In addition to bringing up new challenges for classic kinds of adaptation, such as self-healing, ubicomp stimulates thinking about new kinds of adaptation such as design meshing.

Design meshing is motivated by the increasing expectation that smart spaces make their services available to all occupants, even if at different levels of access. Meshing supports the dynamic deployment and withdrawal of system features as users come and go. Cross effects, and possible conflicts, arise from the sharing of cyber-physical services, such as thermostats, on which operations are expected to have a lasting effect. Conflicts are resolved by associating stakeholder-defined policies with each service provider covering *authorization*, usage *multiplexing* (shared vs. exclusive) and *resolution*, (e.g. priority vs. least misery).

Pliable apps are motivated by context awareness, or more broadly by the need to describe changes in features that are triggered by domain/application events, and that are translated into modes of operation where the application exhibits distinct features. Because the application logic cannot be fully understood without grasping the changes that take place and what causes them, adaptation is defined jointly with other structural and behavioral aspects of the application. A key aspect of pliability is that users may tailor such adaptation and they may change it at run time, depending on the evolution of their needs.

The recast of self-healing into *bond* self-healing, or *B*self-healing, is motivated by harsh environments where *any* component may become disconnected and by the limitations of the devices that typically support ubicomp systems. This pursuit of resilience leads away from centralized components dedicated to self-adaptation and pushes responsibilities towards system components. This does not mean that self-adaptation is being pushed out of middleware, but it does mean that each component is given neighborhood responsibilities, as opposed to having one (or more) component(s) with system-wide responsibilities.

The adoption of a decentralized style to implement self-adaptation has a significant impact on all related mechanisms, from service discovery to the way structural changes are coordinated. This paper discusses implementation strategies that are applicable to approaches for ubicomp self-adaptation other than TeC that share with it the basic tenets of service orientation and of automated deployment of models.

A key point of this paper is that users, both domain experts and end users, will play an increasing role in designing and evolving ubicomp systems. Because some kinds of self-adaptation, e.g. context awareness and multi-user aspects, are a central element of the behavior of such systems, self-adaptation cannot be hidden from users. The language constructs propose here enable users to tailor self-adaptation to their needs, and to evolve it after deployment in response to unanticipated changes.

The evaluation of TeC has so far covered (a) its power to express and deploy systems [6], [18], (b) its amenability to automated checking of behaviors [17], and (c) the usability of the language and editors by users other than the researchers [6].

Ongoing work includes taking the implementation of the adaptation mechanisms discussed in this paper beyond proofs of concept, and incorporating robust versions thereof into the TeC middleware.

This will enable the evaluation of such adaptation mechanisms concerning (i) the effectiveness of adaptation in realistic situations, and (ii) the usability of tailoring adaptation for a diverse user population. Evaluation will proceed in stages: first, in-the-lab empirical studies with college students, starting with science majors and then widening up to the student population at large. Second, empirical studies conducted at the homes of select members of my research group. This will enable creating conditions close to in-the-wild, but where a researcher is close at hand to address possible problems. And third, in-the-wild empirical studies at homes in Masonvale, GMU's housing neighborhood for employees, graduate and professional students.

Acknowledgments. This work was supported in part by grant CCF-0820060 from the National Science Foundation. The views, findings, and conclusions expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] W3C, Web Services Description Language (WSDL 2.0), http://www.w3.org/TR/wsdl20-primer/ (accessed: February 15, 2012)

[2] Ganek, A.G., Corbi, T.A.: The dawning of the autonomic computing era. IBM Systems Journal 42(1), 5–18 (2003)

[3] Zhu, F., Mutka, M., Ni, L.: Service Discovery in Pervasive Computing Environments. IEEE Pervasive Computing 4(4), 81–90 (2005)

[4] Sousa, J.P., Schmerl, B., Steenkiste, P., Garlan, D.: Activity-oriented Computing. In: Advances in Ubiquitous Computing: Future Paradigms and Directions, pp. 280–315. IGI Publishing (2008)

[5] Menascé, D., Gomaa, H., Malek, S., Sousa, J.P.: SASSY: A Framework for Self-Architecting Service-Oriented Systems. IEEE Software 28(6), 78–85 (2011)

[6] Sousa, J.P., Keathley, D., Le, M., Pham, L., Ryan, D., Rohira, S., Tryon, S., Williamson, S.: TeC: End-User Development of Software Systems for Smart Spaces. Intl Journal of Space-Based and Situated Computing 1(4), 257–269 (2011)

[7] Humble, J., Crabtree, A., Hemmings, T., Akesson, K.P., Koleva, B., Rodden, T., Hansson, P.: Playing with the Bits: User-Configuration of Ubiquitous Domestic Environments. In: 5th Intl Conf. Ubiquitous Computing, Seattle, WA, pp. 256–263 (2003)

[8] Truong, K.N., Huang, E.M., Abowd, G.D.: CAMP: A Magnetic Poetry Interface for End-User Programming of Capture Applications for the Home. In: Davies, N., Mynatt, E.D., Siio, I. (eds.) UbiComp 2004. LNCS, vol. 3205, pp. 143–160. Springer, Heidelberg (2004)

[9] Kawsar, F., Nakajima, T., Fujinami, K.: Deploy Spontaneously: Supporting End-Users in Building and Enhancing a Smart Home. In: 10th Intl. Conf. Ubiquitous Computing, Seoul, Korea, vol. 344, pp. 282–291(2008)

[10] Singh, R., Bhargava, P., Kain, S.: State of the art smart spaces: application models and software infrastructure. ACM Ubiquity 7(37) (September 2006)

[11] Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: architecture-based self-adaptation with reusable infrastructure. IEEE Computer 37(10), 46–54 (2004)

[12] Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. In: Future of Software Engineering, Minneapolis, MN, pp. 259–268 (2007)

[13] Georgiadis, I., Magee, J., Kramer, J.: Self-Organising Software Architectures for Distributed Systems. In: Workshop on Self-healing Systems, Charleston, SC, pp. 33–38 (2002)

[14] Sousa, J.P., Poladian, V., Garlan, D., Schmerl, B., Shaw, M.: Task-based Adaptation for Ubiquitous Computing. IEEE Trans on Systems, Man, and Cybernetics, Part C, Sp Issue on Eng. Autonomic Systems 36(3), 328–340 (2006)

[15] Varian, H.R.: Economic Mechanism Design for Computerized Agents. In: USENIX Workshop Electronic Commerce, pp. 13–21 (1995)

[16] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond. Addison-Wesley Professional (2003)

[17] Sousa, J.P.: Foundations of Team Computing: Enabling End Users to Assemble Software for Ubiquitous Computing. In: Intl. Conf. on Complex, Intelligent and Software Intensive Systems, Krakow, Poland, pp. 9–16 (2010)

[18] Sousa, J.P., Tzeremes, V., El-Masri, A.: Space-Aware TeC: End-User Development of Safety and Control Systems for Smart Spaces. In: IEEE Intl. Conf. on Systems, Man, and Cybernetics, Istanbul, Turkey, pp. 2914–2921 (2010)

[19] Sabzevar, A., Sousa, J.P.: Authentication, Authorization, and Auditing for Ubiquitous Computing: a Survey and Vision. Intl Journal of Space-Based and Situated Computing 1(1), 59–67 (2011)

[20] Asmidar, R., Jais, J.: A review on extended role based access control (E-RBAC) model in pervasive computing environment. In: 1st Intl Conf Networked Digital Technologies, Ostrava, Czech Republic, pp. 533–535 (2009)

[21] Baldauf, M., Dustdar, S., Rosenberg, F.: A survey on context-aware systems. Intl Journal of Ad Hoc and Ubiquitous Computing 2(4), 263–277 (2007)

[22] Barkhuus, L., Dey, A.K.: Is Context-Aware Computing Taking Control away from the User? Three Levels of Interactivity Examined. In: Dey, A.K., Schmidt, A., McCarthy, J.F. (eds.) UbiComp 2003. LNCS, vol. 2864, pp. 149–156. Springer, Heidelberg (2003)

[23] Becker, C., Handte, M., Schiele, G., Rothermel, K.: PCOM - a component system for pervasive computing. In: Conf. on Pervasive Computing and Comms, Orlando, FL, pp. 67–76 (2004)

[24] Poladian, V., Sousa, J.P., Garlan, D., Shaw, M.: Dynamic Configuration of Resource-Aware Services. In: 26th International Conference on Software Engineering, Edinburgh, UK, pp. 604–613 (2004)

[25] Sousa, J.P., Zengin, Z., Malek, S.: Towards Multi-Design of Situated Service-Oriented Systems. In: Intl Workshop on Principles of Engineering Service Oriented Systems, Cape Town, South Africa, pp. 57–63 (2010)

[26] Ludford, P.J., Frankowski, D., Reily, K., Wilms, K., Terveen, L.: Because I Carry My Cell Phone Anyway: Functional Location-Based Reminder Applications. In: SIGCHI Conference on Human Factors in Computing Systems, Montréal, Canada, pp. 889–898 (2006)

[27] Gershman, A., McCarthy, J., Fano, A.: Situated Computing: Bridging the Gap between Intention and Action. In: 3rd Intl Symp. on Wearable Computing, San Francisco, CA (1999)

[28] LaMarca, A., Brunette, W., Koizumi, D., Lease, M., Sigurdsson, S.B., Sikorski, K., Fox, D., Borriello, G.: PlantCare: An Investigation in Practical Ubiquitous Systems. In: Borriello, G., Holmquist, L.E. (eds.) UbiComp 2002. LNCS, vol. 2498, p. 316. Springer, Heidelberg (2002)

[29] Bardram, J.E.: Applications of context-aware computing in hospital work: examples and design principles, pp. 1574–1579 (2004)

[30] Mitchell, J.: El Camino Hospital first 'smart hospital' in US. Action for Better Healthcare (2010), http://actionforbetterhealthcare.com/?p=1172 (accessed: February 15, 2012)

[31] Huhns, M., Holderfield, V., Gutierrez, R.: Robust Software Via Agent-Based Redundancy. In: Intl Joint Conf. Autonomous Agents & Multiagent Systems, Melbourne, Australia (2003)

[32] Ghosh, D., Sharman, R., Rao, H., Upadhyaya, S.: Self-healing systems — survey and synthesis. Decision Support Systems 42(4), 2164–2185 (2007)

[33] Aldrich, J., Sazawal, V., Chambers, C., Notkin, D.: Architecture-Centric Programming for Adaptive Systems. In: Workshop on Self-Healing Systems, Charleston, SC, pp. 96–98 (2002)

[34] Cai, H., Shao, Z., Vaynberg, A.: Certified self-modifying code. In: Conf. on Programming Language Design and Implementation, pp. 66–77 (2007)

[35] Oreizy, P., Gorlick, M., Taylor, R., Heimigner, D., Gregory, J., Medvidovic, N., Quilici, A., Rosenblum, D., Wolf, A.: An Architecture-Based Approach to Self-Adaptive Software. IEEE Intelligent Systems (1999)

[36] Rashid, A., Aksit, M. (eds.): Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880. Springer, Heidelberg (2006)

[37] Cetina, C., Fons, J., Pelechano, V.: Applying Software Product Lines to Build Autonomic Pervasive Systems. In: 12th Intl Software Product Line Conf., pp. 117–126 (2008)

[38] Milgrom, P.: Auctions and Bidding: a primer. Journal of Economic Perspectives 3(3), 3–22 (1989)

[39] Fatima, S., Wooldridge, M., Jennings, N.R.: An agenda-based framework for multi-issue negotiation. Artificial Intelligence 152(1), 1–45 (2004)

Hierarchical Self-Optimization of SaaS Applications in Clouds[⋆]

Bradley Simmons[1], Hamoun Ghanbari[1], Sotirios Liaskos[1],
Marin Litoiu[1], and Gabriel Iszlai[2]

[1] York University, 4700 Keele Street, Toronto, Canada
{bsimmons,liaskos,mlitoiu}@yorku.ca, hamoun@cse.yorku.ca
[2] IBM, Toronto Software Lab., 8200 Warden Avenue, Markham, Canada
giszlai@ca.ibm.com

Abstract. This chapter introduces a framework and a methodology to manage a SaaS application on top of a PaaS infrastructure. This framework utilizes PaaS policy sets to implement the SaaS provider's elasticity policy for its application server tier. Adaptation is based on strategy-trees, which allow for systematic capture, representation and reasoning about adaptation variability, based on hierarchically organizing different levels of temporal granularity. Thus, a strategy-tree is utilized at the SaaS layer to actively guide policy set selection at runtime in order to maintain alignment with the SaaS provider's business objectives. This way, the SaaS provider's attitudes and preferences reflecting their general business needs are incorporated into the adaptation mechanism in an organized and accessible manner. Results from an experiment conducted on a real cloud are presented in support of this approach.

1 Introduction

Cloud computing [2, 8, 15, 23] represents an approach to IT which has emerged in large part due to improvements in virtualization technologies[1] [5] and the construction and commoditization of large data centers from which infrastructure (IaaS), platform (PaaS) and software (SaaS) are provided on-demand to end users over the Internet. In fact, this three-layered cloud computing architecture [20] has assumed the role of a *de-facto* standard.

[⋆] This is an extended version of a short paper published previously: B. Simmons, H. Ghanbari, M. Litoiu and G. Iszlai, "Managing a SaaS application in the cloud using PaaS policy sets and a strategy-tree," *Network and Service Management (CNSM), 2011 7th International Conference on*, vol., no., pp.1-5, 24-28 Oct. 2011. "Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than IFIP must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee."

[1] http://www.vmware.com/

R. de Lemos et al. (Eds.): Self-Adaptive Systems, LNCS 7475, pp. 354–375, 2013.

A *PaaS provider* is an enterprise that is responsible for leasing application environment topologies to SaaS provider clients for various durations of time. A topology is built upon infrastructure that is purchased from various IaaS providers upon which the middleware container instances are run. An application environment topology is composed of a set of system service instances S (e.g., load balancers, LDAP servers, ...), platform service instances P (e.g., web server, application server, database server, ...) and the set of licenses L to support all instances in P (should they be required). Additional platform service instances may be purchased from and/or released to the PaaS provider at runtime as needed.

A *SaaS provider* is an enterprise that provides a software offering that is run from within a PaaS topology. There are many possible economic models which can be utilized by a SaaS provider (e.g., free with advertisements, membership cost per period of time, ...).

Regardless of the layer, a service provider has long term business objectives to achieve. Further, as its service is providing some form of IT resources to a dynamic set of clients, how effectively it manages these resources is key to how successful it is in meeting its business objectives. In general, an enterprise will attempt to maximize profit where profit can be understood to represent the difference between revenue and cost.

Consider a SaaS provider running on a cloud. This SaaS provider leases a platform topology from a single PaaS provider and offers one application to a dynamic set of clients. Several aspects of a platform topology (as offered by the PaaS provider) may be configured dynamically via *policy*. A policy can be understood to represent "...any type of formal behavioural guide" that is input to the system [16]. An *elasticity policy* governs how and when resources (e.g., application server instances at the PaaS layer) are added to and/or removed from a cloud environment [13]. One way of specifying an elasticity policy is through a set of policy rules. It has been described previously [30] that a set of policies may be thought of as a strategy. Multiple strategies may be defined to achieve the same set of objectives [31].

It is assumed that the SaaS provider's business objective is to maximize profit. This can involve both the maximization of revenue generation and the minimization of cost. For example, maximization of revenue generation varies directly with the number of clients serviced. Similarly, minimization of cost varies directly with the number of platform service instances that are purchased over time. Different strategies which are defined to achieve the same set of objectives might result in very different outcomes.

Thus, constructing models that allow continuous adaptation of the strategy based on the contextual circumstances (load quantity and quality, prices, service level agreement violation costs) is a challenging problem. One of the problems is the design, comprehension and communication of the adaptation approach which is overly difficult with continuous mathematical models or parametrized policies and without a more systematic modeling approach.

To address this problem of systematically designing the adaptability aspect of software infrastructures, we have introduced the concept of *strategy-trees* [29].

Strategy-trees constitute hierarchical organizations of adaptation decisions, structured in a way that allows stratification of decisions based on the time horizon in which they apply. Thus, long term decisions/options are refined into shorter term ones until they reach rapid low level alternations of fixed configuration strategies. Strategy-trees have the benefit of offering a better organization of adaptation variability, in a way similar to goal models [18, 22] and feature models [26] used in application software engineering, while allowing designers to reason about the depth and cost of adaptation actions.

In this chapter, we present one step toward the realization of our earlier conceptual work with regards to applying strategy trees to a SaaS layer manager in the context of a business driven cloud optimization architecture [19, 29]. The contributions are as follows. We introduce a framework and a methodology to manage a SaaS application on top of a PaaS provider's infrastructure. This framework utilizes PaaS policy sets to implement the SaaS provider's elasticity policy for its application server tier. A strategy-tree is utilized at the SaaS layer to actively guide policy set selection at runtime in order to maintain alignment with the SaaS provider's business objective, specifically to *maximize profit*. Experimental results are presented that reflect positively on this approach.

The remainder of the chapter is structured as follows. Section 2 introduces a scenario involving a SaaS provider running on a PaaS topology which is used throughout the remainder of the chapter. Section 3 provides a brief overview of the concept of strategy-trees and describes the design of a simple strategy-tree for the scenario introduced in the previous section. Section 4 introduces the management architecture for managing SaaS applications on top of a PaaS provider's infrastructure. Section 5 presents an experiment demonstrating the effectiveness of this approach in the context of the introduced scenario. Section 6 offers a discussion of the experimental results. Section 7 provides an overview of some related work. Section 8 presents our conclusions and thoughts on future work.

2 Scenario

Consider a SaaS provider offering a standard multi-tiered application to a dynamically growing and shrinking set of clients. Revenue is proportional to the number of users (sessions) that utilize the service (as each user is statistically linked to some amount of advertising dollars). Cost is impacted by the (i) cost of purchasing the topology and (ii) additional platform service instances purchase over time. There is also a (subjective) cost associated with the loss of future business which is a more speculative (and varies with client response time).

The objective of this SaaS provider is to maximize profit by both maximizing revenue and by minimizing cost. It should be noted that maximizing revenue can have an adverse effect on minimizing cost and vice versa. This interrelatedness greatly complicates the achievement of the main objective. Trade-offs must be made in the pursuit of the overall objective. The next section will consider the design of three, alternative elasticity policies (policy sets), to achieve the objectives under different sets of expectations and assumptions.

2.1 Elasticity Policy

An elasticity policy governs how and when resources are added to and/or removed from a cloud environment. In a production setting, the elasticity policy might be highly complex in order to handle the numerous eventualities and situations that are likely to arise. However, in this illustrative scenario, several simplifying assumptions have been made in order to streamline and focus the discussion.

It is assumed that the SaaS offering (i.e., application) is tightly cpu-bound. This assumption allows us to focus on the single metric, `cpu_idle`, which is considered exclusively in the design of the policy set for this scenario. Further, the policy sets defining the elasticity policy focus only on the application server tier of the SaaS offering. In reality, an elasticity policy is meant to govern changes in resource allocation to all tiers of an application and this may (and would likely) involve the consideration of various application specific QoS metrics as well.

A brief overview of the hierarchical[2], heuristic elasticity policy, utilized in this work, will now be presented (for a more complete overview please refer to [13]). As mentioned already, the policy rules utilized in this work focus on the value of a single performance metric (i.e., `cpu_idle`). When this value is high the implication is that an instance is not being heavily loaded[3]. From a high-level perspective, decisions to grow or shrink the application server tier are made based on a critical number of local observations triggering a global action (i.e., add/remove instances to the application tier). This will now be considered in more detail.

Rules for Platform Service Instances. The policy rules defined for each platform service instance member node of the application server tier of the SaaS provider, introduced for the scenario above, are based on the definition of an acceptable range for the `cpu_idle` metric and an acceptable duration beyond which a violation should be indicated to the management framework. For the remainder of this chapter, we will refer to the upper threshold as `cpu_idle_st` (i.e., the threshold indicating a need to shrink the tier) and `cpu_idle_gt` (i.e., the threshold indicating a need to grow the tier) and the durations will be referred to as `shrink_duration` and `grow_duration` respectively. Some details to consider in relation to the action (i.e., add/remove instances to the application server tier) are as follows. The selection of the *range* (i.e., `cput_idle_gt - cpu_idle_st`) will directly impact the addition and removal of server instances to the tier. Consider if the operating range of the system is well outside of this defined range then violations will consistently occur. In contrast, assuming a more accurate

[2] It is hierarchical in the sense that there are rules that are specified for individual platform service instance nodes in the application server tier and then there are rules that are specified to govern the application server tier that use the results of these lower level rules in aggregate to guide the addition and/or removal of additional platform service instances.

[3] Alternatively, when this value is low the implication is that an instance is being heavily loaded.

prediction of workload and hence a well defined range (i.e., typical operating range is within the threshold values) it will then be the size of the range that will have an impact. Specifically, the size of the range will define how sensitive it is to variation in the workload. Further, the choice of durations will also impact this sensitivity. Specifically, shorter durations will result in more frequent notification to the management system while longer durations will have the opposite effect.

Rules for the Application Server Tier. The policy rules defined for the application server tier of the SaaS provider, introduced in the scenario above, involve several configurable parameters as well. First is the definition of the value of a `quorum`. A quorum denotes the percentage of instances that must all be indicating that they are in violation of their local policy rule (all members of the quorum must indicate the same violation e.g., above `cpu_idle_st` or below `cpu_idle_gt`) in order to trigger an auto-scaling action (i.e., grow or shrink the tier). The amount by which the tier is meant to be grown is indicated by the parameter `incr_val`. The amount by which the tier is meant to be shrunk is indicated by the parameter `decr_val`. Whether an auto-scaling action even occurs is controlled by the parameter `refractory_period` which indicates the amount of time that must have elapsed since the last auto-scaling action took place. The way these parameters impact the auto-scaling behaviour of the tier is as follows.

The values `incr_val` (or `decr_val`) affect how aggressively the grow/shrink action will be. Specifically, a larger number indicates a more aggressive action. The value of `quorum` impacts the sensitivity of individual indications of a violation. A larger quorum (i.e. closer to 100%) implies more sensitivity to individual notifications while a smaller quorum implies the opposite. Finally, the larger values of `refractory_period` will result in a more gradual change in tier size while a smaller value will have the opposite affect.

An example of the policy rules defining the auto-scale grow action are provided in Listing 1.1. For the remainder of this document, an elasticity policy is defined by a set of four policy rules (two for growing and two for shrinking). Three different elasticity policies were designed to drive the auto-scaling actions of the application server tier under different circumstances. These policy sets utilized different settings of some of the configurable parameters mentioned above and are presented in Table 1. The first elasticity policy, $P_{Sensitive}$, was designed to be gentle in how it grew/shrunk the tier (i.e., adding/removing only one platform service instance at a time). Similarly, the second elasticity policy, $P_{Tolerant}$, increased and decreased the topology in a gentle fashion as well; however, the range separating its two thresholds (upper and lower) was three times as large as for $P_{Sensitive}$ making it much less likely to be triggered as often (assuming violations occur inside the defined range). The third elasticity policy, $P_{Aggressive}$, was designed to be much more aggressive in how it grew/shrunk the tier as evidenced by both a small range between its upper and lower thresholds and an increment value (up and down) of two.

```
(a) inst oblig cpu_idle_breach_low {
      subject s = inst_mgr;
      target t = platform_tier_mgr;
      on {e1 ; e2} ! e3
      do emit(t, request_increase)
      when e2.time - e1.time == grow_duration and
        e1.cpu_idle < cpu_idle_gt and
        e2.cpu_idle < cpu_idle_gt and
    }//cpu_idle_breach_low
(b) inst oblig perform_autoscale_grow {
      subject s = paas_mgr;
      target t = platform_tier_mgr;
      on  quorum(platform_tid, action)
      do  t.elastic_grow_action(incr_val)
      when action.equals("grow") and
        !t.refractory_period and
        t.id == platform_tid
    }//perform_autoscale_grow
```

Listing 1.1. Sample policies to auto-scale grow the platform tier specified in a Ponder-like [11] notation. In (a) it is assumed that e3 denotes an event indicating `e3.cpu_idle` ≥ `cpu_idle_gt`.

Table 1. Parameter settings defining the three elasticity policies as used in the experiments (i.e., values related to time are scaled by one quarter)

Parameter	$P_{Sensitive}$	$P_{Tolerant}$	$P_{Aggressive}$
incr_val	1	1	2
decr_val	1	1	2
quorum	51%	51%	51%
cpu_idle_gt	45	40	50
grow_duration	7 min	7 min	8 min
cpu_idle_st	50	55	55
shrink_duration	7 min	7 min	8 min
refractory_period	8 min	8 min	6 min

3 Strategy-Trees

A *strategy* can be defined as "...a plan of action designed to achieve a long-term or overall aim"[4]. In the context of policy-based management, a set of policies can be understood to implement a strategy. The *strategy-tree* was introduced to address a deficiency in current approaches to distributed system's management. Simply put, there can exist multiple strategies to achieve a *directive* (i.e., a set of objectives[5]). These alternative strategies often incorporate assumptions, biases and expectations within a given policy set (i.e., the management logic which governs the

[4] http://oxforddictionaries.com/definition/strategy

[5] An objective represents a constraint on a metric. A metric might be a low level technical metric (e.g. throughput) or a business metric (e.g., profit).

system's behaviour attempting to achieve the set of objectives). Under different scenarios various assumptions can be more/less correct than others resulting in different degrees of effectiveness for the various strategies. Through monitoring of the progression toward the system's objectives and by utilizing feedback about the effectiveness of the deployed strategy (with regards to achieving the defined objectives) informed decisions can be made allowing an ineffective strategy to be changed to an alternative, to better meet the long term objectives.

The concept of a strategy-tree was introduced to facilitate intelligent switching among defined policy sets (i.e. strategies) at runtime in response to monitored data and in the pursuit of a a directive defined over a long-term horizon of time. In essence, a strategy-tree represents a framework for reasoning about the effectiveness of an active strategy. In this sense it is a tool for meta-policy management [11]. While everything it accomplishes, might possibly be done using a set of highly complex and convoluted policies, this abstraction simplifies and organizes the process of evaluating the effectiveness of a deployed policy set and orchestrates the switching among alternative strategies over time in a defined, systematic and hierarchical manner. Further, this approach provides an architecture to facilitate this process of strategic management[6]. For a more comprehensive and formal consideration of strategy-trees and their use in policy management please refer to [28–31]; however, a brief description of the key points follows.

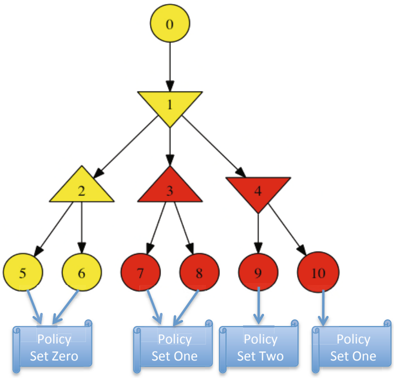
A strategy-tree, Fig. 1, is composed of three types of nodes: Directive (i.e., circle), AND (i.e., triangle) and OR (i.e., inverted triangle). Associated with each node in the tree is a *quantum attribute value* which denotes when[7] a node's (that is a member of the active strategy) SAT-element [8] should be executed and in the case of an OR type node its DEC-element as well. There is also a list, *results*, that is associated with each node which enables a child node to pass up the result of its evaluation (i.e., execution of its associated SAT-element) to its parent node (for use in later evaluations). Each leaf node of a strategy-tree is bound to a single policy set[9].

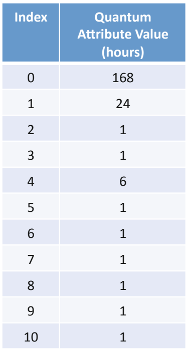
[6] Strategy-trees are not meant to handle asynchronous problems. Changes in strategy are gradual and occur on scales of hours, days, weeks, months, years, etc. (not milliseconds). It is assumed that for gross, pathological errors there are policies defined to handle these situations. There is also overhead associated with deploying policy sets and this should not be ignored.

[7] A quantum attribute value represents a coefficient on some management time unit (MTU).

[8] SAT-elements are used to evaluate whether a set of objectives is satisfied. DEC-elements are used in decision making to determine whether to maintain the current strategy or to switch to an alternative.

[9] While multiple leaf nodes may be bound to the same policy set, leaf nodes of different strategies may only do so under certain constraints. The policy set P_a and P_b that are bound by two leaf nodes that are both direct child nodes of the same OR type node must not be equivalent. Further, should two leaf nodes have a Lowest Common Ancestor (LCA) that is an OR type node and should there be no intervening OR type nodes between either leaf node and this OR type node then the policy sets, P_a and P_b, bound by these two leaf nodes must not be equivalent either [28].





Index	Quantum Attribute Value (hours)
0	168
1	24
2	1
3	1
4	6
5	1
6	1
7	1
8	1
9	1
10	1

Fig. 1. An example strategy-tree with 11 nodes. There are four strategies $S_0 = (0, 1, 2, 5, 6)$, $S_1 = (0, 1, 3, 7, 8)$, $S_2 = (0, 1, 4, 9)$ and $S_3 = (0, 1, 4, 10)$. Currently, S_0 is active as denoted by its yellow coloring (red indicates inactive). The quantum attribute values presented in the table are determined through experimentation, simulation or may even be selected arbitrarily.

At a high level, the algorithm for evaluating a strategy-tree goes as follows[10]. Each time the strategy-tree is evaluated (i.e., each increment) the SAT-elements in the active strategy are processed from leaf to root. A SAT-element is evaluated when the increment value modulo the node's quantum attribute value is equal to zero. Once all the SAT-elements have been evaluated the DEC-elements of the active strategy are evaluated from root to leaf[11]. Should any DEC-element decide to switch strategy, the switch is implemented and the algorithm terminates. So, if we assume the strategy-tree in Fig. 1, and further that S_0 is the active strategy then every hour the SAT-elements associated with nodes five and six evaluate and pass up results to node two which aggregates[12] this result and passes it up to node one. At iteration 24, after the twenty-fourth evaluation of these SAT-elements, the SAT-element associated with node one evaluates and passes its result up to node zero. Next, the DEC-element associated with node one is evaluated and a decision, based on the most recent epoch (i.e., 24 hours) of collected data, is used to decide whether to continue using strategy S_0 or whether to switch to one of the three alternatives (i.e., S_1, S_2 or S_3).

A strategy-tree that has multiple OR type nodes, as in Fig. 1, can be understood to have multiple MAPE loops [17] defined. For example, node four, implements a loop that uses the six most recent results for the evaluation of SAT-elements associated with node nine or ten (depending on whether strategy S_2 or S_3 is active) as well as all monitored data for this six hour period in its decision making process. In contrast, node one, implements a loop which

[10] This assumes that the tree has been fully specified, all elements defined, all leaf nodes have been bound to policy sets, and the initial strategy set to active.

[11] This is a small, yet valuable (in terms of complexity) alteration to the algorithm.

[12] Applies a *boolean* AND to the results.

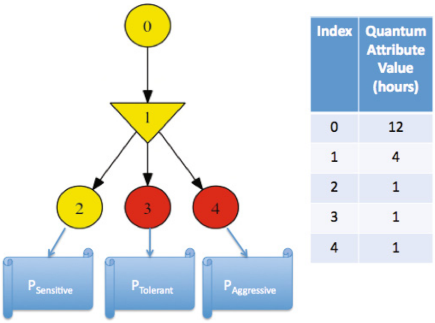
utilizes the previous 24 results for the evaluations of SAT elements associated with nodes five, six and two (when strategy S_0 is active) or for the evaluations of SAT-elements associated with nodes seven, eight and three (when strategy S_1 is active) or the four most recent evaluations of the SAT-element associated with node four (when strategy S_2 or S_3 is active) as well as all monitored data for this 24 hour period in its decision making process. It should be pointed out that in all but the simplest cases, more data than just the previous epoch's[13] will be used in the decision making at a DEC-element.

3.1 Scenario: Designing a Strategy-Tree for the SaaS Provider

This section considers the development of a strategy-tree, Fig. 2, to help guide the system to achieve the objective of the SaaS provider (i.e., maximize profit) introduced in Section 2. Recall from Section 2.1 that three elasticity policies have been defined: $P_{Sensitive}$, $P_{Tolerant}$ and $P_{Aggressive}$. Each of these elasticity policies (policy sets) can be viewed as a particular strategy to achieve the SaaS provider's objectives under a particular set of expectations and assumptions.

The strategy-tree that we will consider consists of only five elements: four directive type nodes and one OR type node. This simple structure was intentionally selected in order to focus on the development of the single DEC-element for the OR type node. To achieve the objective, heuristic trade-offs between maximizing revenue and minimizing costs are utilized. A bias which favours servicing the maximum number of clients while attempting to limit the number of additional platform service instances purchased is applied.

Characterizing the Elasticity Policies. The various strategies need to be understood in order to evaluate their effectiveness and reason about switching among possible alternatives. The SaaS provider was able to characterize each of the three (see Section 2.1 and Table 1) elasticity policies against a standard workload (i.e., trace data that they had access to). For each policy set,



Index	Quantum Attribute Value (hours)
0	12
1	4
2	1
3	1
4	1

Fig. 2. Strategy-tree used for the experiment

[13] An epoch is equivalent to the quantum attribute value of the node in question so if a node has a quantum attribute value of 24 hours then the epoch is 24 hours as well.

the mean hourly number of additional platform service instances purchased was computed. They were also able to monitor the current number of sessions at four minute intervals. On this data, they performed hourly regressions and partitioned the slopes of these regressions into four distinct categories (indicating different degrees of increase/decrease in numbers of sessions).

Design of SAT-Elements. The three leaf nodes (e.g., nodes two, three and four) each have quantum attribute values of one hour. This implies that each hour, they evaluate their SAT-element and pass the result up to node one. Node one evaluates its SAT-element every four hours and passes its result up to node zero. Each of these elements is evaluating the following objective:

– *The number of additional platform service instances purchased, divided by the epoch, should not exceed the hourly mean for that particular strategy.*

Design of the DEC-Element. The design of the DEC-element for node one was much more involved than for the SAT-elements in the tree. This is normal as deciding among alternative approaches can be difficult at the best of times.

In order to guide performance toward achieving the objective (i.e., maximize profit) it was decided that a two step approach would be utilized when deciding whether to continue using a particular strategy or whether to switch to an alternative. This decision would be based first upon the detection (or lack of detection) of a trend in the number of current sessions observed over the previous epoch. Specifically, the slope of the hourly regressions (for the previous four hour epoch) constructed from the readings (i.e., current number of sessions) taken every four minutes would provide a simple heuristic for detecting a rapid increase or decrease in the client demand on the system (and hence guide the decision making process to use the more aggressive strategy). Should no strong trend be detected, data from the MDB as indicated by the values from the *results* list (about the additional purchased platform service instances) for the previous four hour epoch would then be utilized.

The logic underpinning the DEC-element works as follows. Every increment of the MTU[14], prior to evaluation of the strategy-tree, data about the application is collected and stored in the MDB. Specifically, it includes the previous 15 current session readings and the 15 additional purchased platform service instance readings as well. A regression is performed on the set of current session readings and this is then stored for later use.

When the DEC-element at node one is executed and the method `decider`[15] is invoked, the following steps occur. First, the four most recent regressions are collected from the MDB. Next, the previous 60 additional platform service instance purchases are collected and summed. This data in combination with the current active strategy denotes the *context.* Regardless of whether the *active* strategy is $S_0 = (0,1,2), S_1 = (0,1,3)$ or $S_2 = (0,1,4)$, Fig. 2, a call is made to

[14] MTU refers to the management time unit which in the case of this scenario is 60 minutes.

[15] This is the name of the method which evaluates the DEC-element's decision problem.

the method, `map_degrees_to_range`. This method accepts an array of the four most recent hourly regressions $R = [r_1, r_2, r_3, r_4]$. From each regression r_i the slope m_i is extracted and an integer value returned denoting membership in one of the four defined categories:

(i.) $0° < m \leq 85° \mapsto 1$
(ii.) $m > 85° \mapsto 3$
(iii.) $0° > m \geq -85° \mapsto -1$
(iv.) $m < -85° \mapsto -3$

This resultant array of integer values $I = [i_1, i_2, i_3, i_4]$ is multiplied by an array of weights $W = [w_1, w_2, w_3, w_4]$ where $w_1 = 1$, $w_2 = 2$, $w_3 = 4$ and $w_4 = 8$. Notice that the weight values in this array are increasing which ensures that emphasis is placed on the more recent regression's slope. The summation of the multiplication of $W * I$ is returned: $\Sigma_{i=0}^{n} W[i]I[i]$.

Two trends were identified as being of interest: steeply increasing (SI) and steeply decreasing (SD). Specifically, any sum $\geq$ 27 is considered to be SI while any sum $\leq$ -27 is considered to be SD. All other values are considered to be less indicative of a trend (either increasing or decreasing) and this is when questions about the number of purchased platform service instances are considered.

If the current active strategy is S_0 the following reasoning is used. If the value of *sum* as returned by the call to `map_degrees_to_range` indicates *SD* or *SI* then the variable *next_strategy* is set to two. The rationale for this choice is that if a steep increase or decrease in the number of sessions is observed, it is important (according to the preferences of the SaaS provider) to respond to this by using the aggressive policy set $P_{Aggressive}$ (i.e., S_2) in order to ensure that available resources are allocated/de-allocated to handle the sharp change in demand. Otherwise (i.e., it is not SD or SI), if the mean number of platform service instances purchased (over the previous four hours) is greater than the mean for S_0 (recall that this is equivalent to $P_{Sensitive}$) then set *next_strategy* to two (indicating S_2) otherwise, set it to one (indicating S_1).

If the current active strategy is S_1 the following reasoning is used. For the cases where *sum* indicates either *SD* or *SI* the identical reasoning as for S_0 is used. However, as it is now $P_{Tolerant}$ being employed if no apparent trend is perceived and if the mean number of platform service instances purchased (for the four hour epoch) is greater than the mean for this strategy then *next_strategy* is set to zero otherwise it is set to one. A simple rationale is used in deciding the switch to S_0. It had been determined during characterization of the three elasticity policies (see Section 2.1) that the hourly mean number of additional platform service instances purchased is lower under S_0 than S_1. Since no trend has been observed (i.e., not SD or SI), and the mean number of platform service instances purchased over the previous epoch has exceeded the mean for S_1 by switching to S_0 there will be fewer additional platform service instances purchased in the next epoch. It is hoped that this will apply a downward pressure on the overall number of additional platform service instances purchased (i.e., decrease the cost). Similarly, if the current active strategy is S_2 the same reasoning as for S_1 is used, except, the threshold for S_2 is substituted in place of the threshold for S_1.

4 Architecture

The following section will provide an overview of our proposed management architecture, Fig. 3. Since the scenario we are considering involves a SaaS provider running an application on top of a PaaS topology we focus only on these two layers.

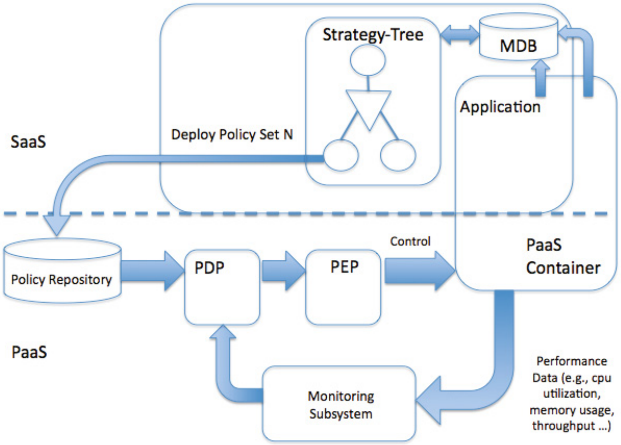


Fig. 3. Proposed management architecture

4.1 PaaS Layer

At the PaaS layer, we assume the traditional policy-based management architecture (PBM) consisting of Policy Repository, Policy Decision Points (PDP) and Policy Enforcement Points (PEP) [21]. While the SaaS provider may also utilize PBM internally, we leave this undefined for now as we are only considering PaaS layer policy sets in this work and their management by a strategy-tree in order to achieve its root directive.

The PaaS provider has access, via a monitoring subsystem, to numerous performance metrics (e.g., cpu utilization, throughput, etc). It also has access to various OS (e.g., ps count, ps cputime, ...) and middleware level metrics (e.g., request queue length, transmitted bytes, session count, ...) as well. We assume that the PaaS provider exposes these metrics to its SaaS clients so that they may define policy rules with which to implement their elasticity policies. Policy rules are specified in the traditional *On-event-If-condition-Then-action* syntax.

4.2 SaaS Layer

A strategy-tree is used at the SaaS layer[16] to dynamically alter policy set deployment at runtime. We assume that it has access to various performance metrics that allow it to determine whether the currently deployed strategy is effective or

[16] It is the SaaS layer manager at this point.

not. For example, it is aware of how many platform service instances it has purchased over time and also how many sessions it has serviced. The management database (MDB) is where this data is stored. All elements of the strategy-tree have access to it.

5 Experiment

The following experiment is based on the scenario of the SaaS provider introduced in Section 2 and is composed of two parts. First, the three elasticity policies (i.e., Table 1) are characterized against a workload as described in Section 3.1. Then the strategy-tree (i.e., Fig. 2) is deployed and each policy set and the strategy-tree are run against a novel workload and compared in terms of total sessions, number of additional platform service instances purchased and mean response time as measured at the client.

5.1 Experimental Setup

For this experiment, Amazon (i.e., EC2, EBS) was used as the IaaS provider. All topology instances were built atop virtual machine instances (VMI)s running either CentOS 5.4 i386 (i.e., front end servers and application server instances) or Ubuntu 8.04 i386 (i.e., database) and configured as m1.small instances (i.e., 1.7 GB memory, 1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit), 160 GB instance storage, 32-bit platform and I/O Performance: Moderate).

RightScale was used as a PaaS management framework. A standard, multi-tiered application topology was selected from their catalog (with various modifications to suit our needs). This platform topology consisted of two front-end servers running Apache and HAProxy an array of Tomcat instances and a back end database running MySQL 5.0. The concepts of elastic scaling of a server array using alerts (based on voting tags) employed by Rightscale allowed us to specify our elasticity policies. We wrote lightweight Policy and PolicySet classes that were implemented in Ruby. Once fully specified, a PolicySet could be deployed (utilizing Rightscale's restful API) at which point it would result in the configuration of the platform topology with the correct elasticity policy as previously described. The strategy-tree was implemented in Ruby and the initial encoding is from within an XML file[17].

The client is run on a separate EC2 instance and simulates the correct number of clients as defined by the workload for the duration of the experiment. The workloads used both to characterize the three elasticity policies and for the actual experiment were excerpts from the FIFA '98 workload [1] (Figs.4a and 5a).

Experiment time was scaled by four The monitoring system at the SaaS provider takes a reading every minute (i.e., four minutes of experiment time). At the SaaS provider layer, a simple Java-based web application was deployed on the described PaaS topology. A client connects to the front-end, is directed to an application

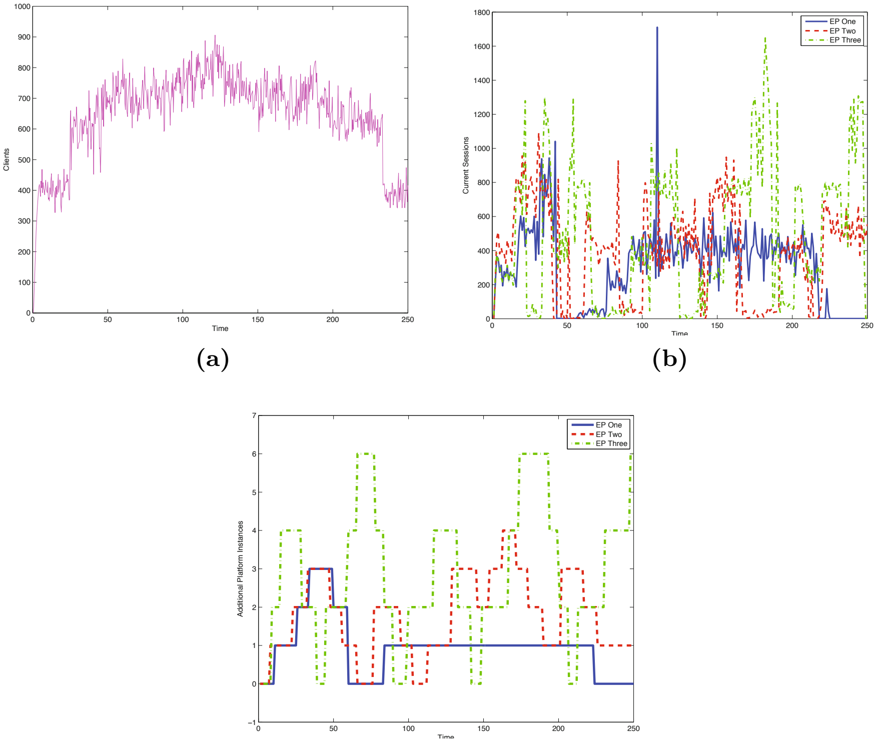
[17] This is an updated version from previous work where it was implemented in Java.

server, a loop executes some pre-defined number of times (i.e., for this work we focused on the CPU) communication with the database tier occurs and a response is issued. For the remainder of the chapter, this will represent a session.

5.2 Characterizing the Elasticity Policies

The first step when using a strategy-tree requires that a characterization of the various strategies be performed. The results of running a single workload (FIFA '98, Day 41, partial excerpt) against the SaaS application utilizing each of the three elasticity policies is presented in Fig. 4. Notice that the differences in both current sessions (Fig. 4b) and additional platform service instances utilized over time (Fig. 4c) varies substantially among the three alternative strategies.

Following each run, the hourly mean number of platform service instances purchased by the SaaS provider when operating under that elasticity policy was computed. Also, hourly regressions were computed on the *number of sessions* [18]



(a)

(b)

(c)

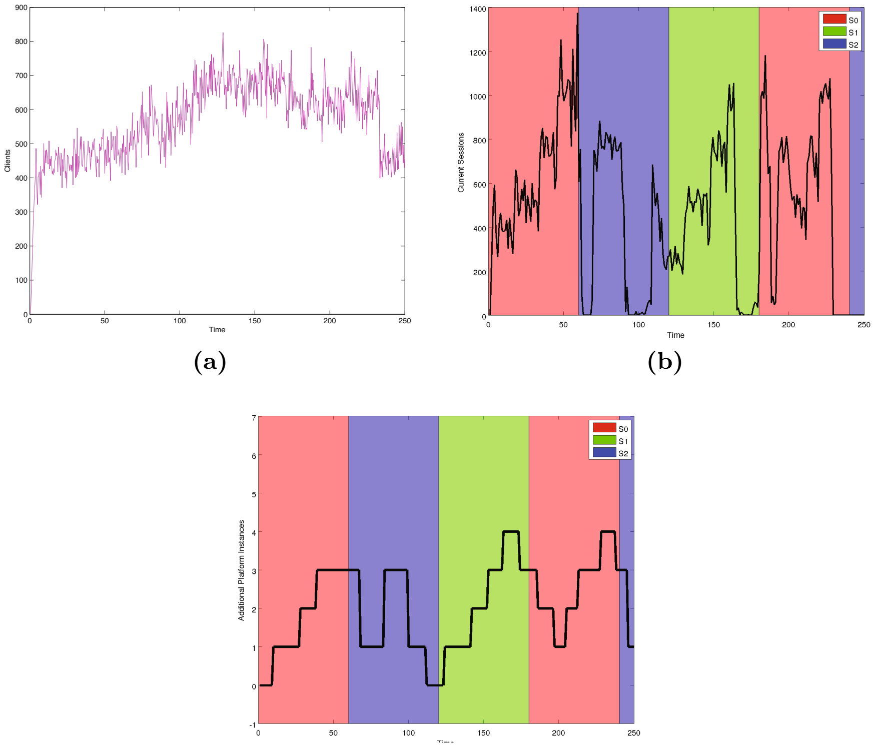
Fig. 4. (a) FIFA '98, Day 41, partial excerpt. (b) Number of sessions processed by the application in response to the workload using three alternative elasticity policies (EP)s. (c) Additional platform service instances being added and released in response to the workload under the three alternative EPs during characterization phase.

[18] There were 15 readings per experiment hour.

serviced by the SaaS offering for each complete run under each elasticity policy. The slopes of these regressions were then partitioned into four categories (i.e., ranges) of roughly equal occurrence as described in Section 3.1. These characterizations were used in the design of elements for the strategy-tree as previously described.

5.3 Experimental Results

An alternative workload (Fig. 5a) was selected (FIFA '98, Day 43, partial excerpt). The workload was pre-processed so as to stretch the y-coordinates by a factor of 1.4 (to increase the number of clients)[19]. We ran three repetitions for each strategy, Table 1, and for the strategy-tree, Fig. 2. Plots from one of the three runs using a strategy-tree are presented in Figs. 5b and 5c. In this run,



(a) **(b)**

(c)

Fig. 5. (a) FIFA '98, Day 43, partial excerpt (stretched by 1.40). (b) Total number of sessions processed versus time: strategy-tree. (c) Platform service instance usage versus time: strategy-tree.

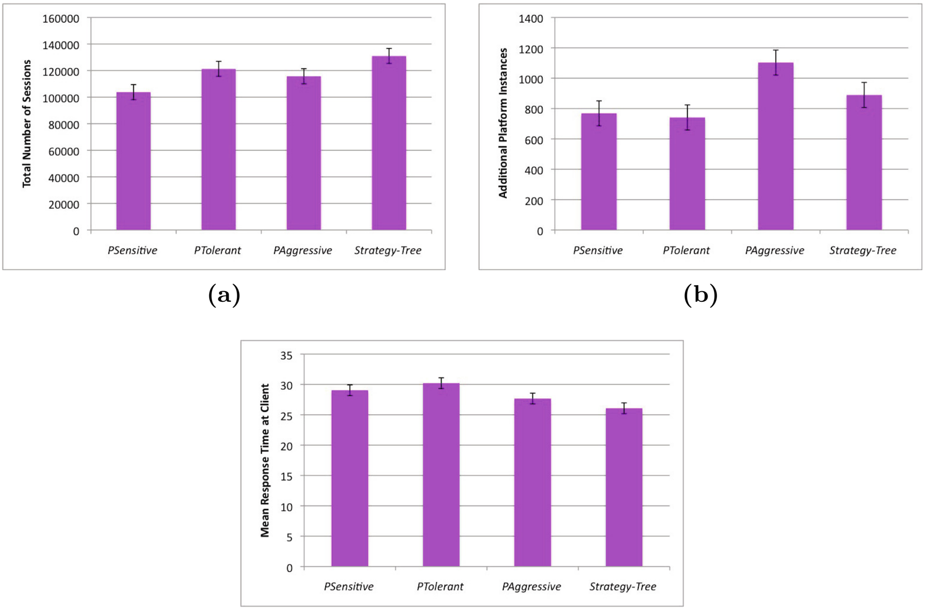
[19] This stretch was applied as the workload did not look very interesting initially (i.e., its maxima were much less than the day 41 partial excerpt data we had initially worked with)

Table 2. Placement for various approaches. Total Sessions (Tot. Ses.), Additional Instances (Add. Inst.), Mean Response Time at the client (MRT), and Strategy-Tree (ST). There are twelve trials. For each row, 1 denotes the best result for that metric and 12 denotes the worst.

Metric	$P_{Sensitive}$	$P_{Tolerant}$	$P_{Aggressive}$	ST
Tot. Ses.	8,11,12	3,4,6	5,9,10	1,2,7
Add. Inst.	4,5,6	1,3,7	9,11,12	2,8,10
MRT	2,10,11	6,8,12	4,5,9	1,3,7

all three strategies were used at various points in time as indicated by the alternative background colourings. An overview of the results for the individual trials is presented in Table 2. This table assigns an integer ranking (i.e., one denotes best while 12 denotes worst) as follows: a strategy is more successful when it services a greater number of total sessions, purchases fewer additional platform service instances and provides a lower mean response time to its clients.

Figure 6a presents the mean total number of sessions serviced for each set of three runs for each approach (i.e., $P_{Sensitive}$, $P_{Tolerant}$, $P_{Aggressive}$ and the strategy-tree). Figure 6b presents the mean number of platform service instances purchased for each set of three runs for each approach. Figure 6c presents the mean of the mean response time at the client for each set of three runs for each approach.



(a) **(b)**

(c)

Fig. 6. Mean of three trials for each elasticity policy and for the strategy-tree (plus and minus one standard error) for (a) Total sessions. (b) Total number of platform service instances. (c) Mean response time as measured at the client.

6 Discussion

The experiment presented in Section 5 was preliminary in both its scope and complexity; however, it demonstrates that a strategy-tree can achieve its root's directive (i.e., maximize profit). Recall from Section 2 that the objective of the SaaS provider is to maximize profit by both maximizing revenue and by minimizing cost. It attempts to achieve this through the strategy-tree which employs a bias (at its DEC-element) that favours servicing the maximum number of clients while attempting to limit the number of additional platform service instances purchased. Finally, recall that their is also an additional speculative cost associated with loss of future business due to sub-par response time.

In terms of best individual results (see Table 2) the strategy-tree approach finished first both in total number of sessions serviced and mean response time at the client. It also finished second for additional platform service instances purchased. The best individual results for $P_{Sensitive}$ were eighth for total number of sessions, fourth for additional platform service instances purchased and second for mean response time at the client. The best individual results for $P_{Tolerant}$ were third for total number of sessions, first for additional number of platform service instances purchased and sixth for mean response time at the client. The best individual results for $P_{Aggressive}$ were fifth for total number of sessions, ninth for additional number of platform service instances purchased and fourth for mean response time at the client. Further, it should be pointed out that the strategy-tree approach never obtained the worst result for any of the metrics while all individual strategies did. Also, its two worst individual results were in terms of the number of additional platform service instances purchased and this can be understood due to the bias in favour of servicing client sessions. These results argue in favour of the effectiveness of the strategy-tree at facilitating trade-offs while maintaining alignment with the root directive.

In terms of aggregate results, the mean value over three trials for the total number of sessions (see Fig. 6a) and the mean value over three trials for mean response time at the client (see Fig. 6c) were better for the approach utilizing a strategy-tree than for any of the individual strategies. Further, the mean value over three trials for number of platform service instances purchased (see Fig. 6b) was much less for the approach utilizing a strategy-tree than for $P_{Aggressive}$. More precisely, the strategy-tree approach serviced approximately 26% more sessions, while using approximately 16% more platform service instances and achieving an improvement in mean response time at the client of approximately 10% when compared to $P_{Sensitive}$. The strategy-tree serviced approximately 8% more sessions[20], while using approximately 20% more platform service instances and achieving an improvement in mean response time at the client of approximately 14% when compared to $P_{Tolerant}$. The strategy-tree serviced approximately 13% more client sessions, while using approximately 19% fewer platform service instances and achieving an improvement in mean response time at the client of approximately 6% when compared to $P_{Aggressive}$.

[20] This is actually quite substantial (i.e., approximately 10,000 sessions).

Various factors may have adversely impacted the results presented here. For example, the set of experiments (both characterizations of the three elasticity policies and the actual experimental runs) were performed on the Internet and not on a private test-bed; further, the experiments were run without regard for time of day and this might have had an impact.

The policy sets that were used were heuristic in nature with no formal methodology utilized in their design. We intend to investigate designing techniques to better determine thresholds for our policy rules, using formal modelling techniques and building on work done in [6]. It should be emphasized that the intent of this chapter was to explore reasoning about the performance of the policy sets via a strategy-tree rather than focusing on the optimal design of a specific elasticity policy. In fact, the elasticity policies used in this chapter were developed in an *ad-hoc* fashion in contrast to the formal refinement approaches such as [4,10,25].

One limitation of the strategy-tree presented here is that it is only reactive in nature. Specifically, it only considers the previous epoch's history. This falls into the problem of local minima/maxima (i.e., hill climbing problem) . One possible approach to improve this limitation would be to utilize the growing history over time. However, to truly implement good decision making in DEC-elements prediction is required. Work by [14] utilized signal processing techniques to detect patterns in workloads to assist in prediction. These forms of techniques would be interesting to apply inside DEC-elements of a strategy-tree.

7 Related Work

The strategy-tree was introduced to allow for changes in strategy (i.e., policy set) to be made in response to changes in experienced workloads and/or failures of the implicit assumptions underpinning policy set construction. Regardless, it is predicated on achieving a long-term directive over some horizon of time based on a pre-defined set of policy sets. The work presented by [3] describes a mechanism for applying reinforcement learning in the context of policy management. In this work, the *active* (i.e., deployed) policy set is utilized to dynamically construct a model of the system which guides the autonomic manager resulting in improved performance. As changes in the deployed policy set occur Bahati et al. are able to demonstrate that in a majority of cases transformations can be performed on the state-transition graph thus retaining much of the previously learned information; however, they also demonstrate that in certain circumstance (i.e. model transformation $\Psi_5[G_n^P]$), the initial model must be discarded and a new one initialized. We see a DEC-element as potentially representing a transformation of this type. Further, the idea of dynamically evolving a policy set is one that we have currently left unexplored but see value in. Presently, a change in policy set implies a change in strategy (at any particular DEC-element). However, allowing for policy sets to be learned may result in a less constrained approach.

The work in [7] introduces the SYMIAN decision support tool which is used to determine effective incident management strategies. Each DEC-element in a strategy-tree denotes a unique locus of control for selecting among a set of strategies. However, unlike SYMIAN, there exists a hierarchy of objectives to achieve in which the effectiveness of employed strategies to achieve lower layer objective sets directly contributes to the perceived achievement of those objective sets higher in the tree structure. Further, unlike SYMIAN, which facilitates strategy design, the decision making at a DEC-element in a strategy-tree is done automatically at runtime over a pre-determined set of alternative strategies. While the DEC-element described in this chapter was designed based on a simple heuristic, a need for more involved reasoning among potential strategies especially as temporal granularities grow and out-degree of DEC-elements (i.e., OR type nodes) increases is clear. Each DEC-element represents a multi-criteria decision problem in which the current context (i.e., monitored data, the current active strategy and the set of available strategies at the particular node) must be considered while trying to achieve a local set of objectives. The decision to maintain or switch current strategies can become a highly complex and challenging problem requiring well thought out models, and approaches. In effect, a strategy-tree with multiple DEC-elements (as in Fig. 1) represents a hierarchical (over time) set of decision problems to be solved.

The work in [24] manages the Quality of Service (QoS) provisioning of Diff-Serv over MPLS networks in alignment with business objectives. A model of business utility is introduced relating business indicators, SLA indicators, objectives and policies. The business indicators are assigned weights and a set of mapping functions are derived to facilitate the generation of effective policy parameters. This approach is an extension to the policy refinement work in [25]. The policies that are considered are simple rules and the weights (while justified) appear somewhat arbitrarily selected. Regardless, the method demonstrated through simulation seems promising. It should be emphasized that once a strategy is selected, that is it. There is no mechanism to change strategy automatically. In contrast, a strategy-tree utilizes feedback at (typically) multiple temporal granularities to allow for changes in strategy to be made over time. One possible avenue would be to determine various weight settings based on sets of assumptions (as presented informally in the paper) and then to construct a strategy-tree to alternate among these alternative strategies over time while maintaining alignment with the long term business objectives.

The Stitch language [9] was introduced to specify adaptation strategies in the context of the Rainbow Framework [12]. Stitch is based on three main concepts: *operators*, *tactics* and *strategies*. An operator maps to a system provided command (i.e., an effector), a tactic represents a conditional evaluation of a set of actions (i.e., calls to operators) and an expected set of effect(s) and a strategy which is a tree of conditional tactic delay nodes[21]. At runtime a strategy is selected from the set of possible strategies based on its overall utility across a set of quality dimensions, d, in a particular context. The sum of utility values is

[21] Associated with each node in a strategy is a probability that its condition will evaluate to true and a delay specifying the horizon over which the effect of the tactic's execution will be observable.

constructed based on weights, w_i, which are arbitrarily defined (e.g., $U = \sum w_d u_d$) as are the utility functions for each dimension. Comparatively, a strategy, in terms of a strategy-tree, denotes the entire set of management policies which are deployed at any given point in time. With strategy-trees we are not trying to manage a single adaptation, rather, we are attempting to guide the system, in a more scalable fashion to achieve the long-term objectives of the administrator. In fact, Stitch and the strategy-tree approach are complementary, as a module of Stitch strategies could be viewed as the deployed policy set while alternative modules of strategies (i.e., ones with different probabilities, weights and utility functions) could be thought of as alternative policy sets and a strategy-tree could then be constructed to switch among these modules at runtime to maintain alignment with the long term management objectives.

8 Conclusions

The work presented in this chapter is an initial step toward the realization of our business driven cloud optimization architecture. We introduced an architecture and methodology for managing a SaaS application on top of a PaaS provider's infrastructure. This framework utilizes PaaS policy sets to implement the SaaS provider's elasticity policy for its application server tier. A strategy-tree is utilized at the SaaS layer to actively guide policy set selection at runtime in order to maintain alignment with the SaaS provider's business objectives. An experiment on a real cloud was presented that demonstrates the promise of this approach and the usefulness of dynamically switching among active strategies at runtime.

In future work we would like to use a more realistic application in which multiple classes of clients are defind and various admission control policies and tuning policies can be used to augment the complexity of the application's elasticity policy. We also intend to continue developing the concept of the strategy-tree. While initially, a strategy-tree was designed to capture implicit objectives underpinning the various policy sets, we think there may also be an interesting research space connecting it to explicit objectives. Specifically, we feel that there may exist a link between the concepts of goal-models, awareness requirements [27] and strategy-trees that might allow us to automate the generation of the tree structure as well as the various SAT-elements so that the strategy-tree is more directly connected to the objectives of the service (e.g., SaaS) provider and easier to build and use. We are also beginning to suspect that perhaps simulation is a better avenue for demonstrating longer-term strategic management than through experimental work.

Acknowledgment. This research was supported by the IBM Centre for Advanced Studies (CAS), the Natural Sciences and Engineering Research Council of Canada (NSERC), Ontario Centre of Excellence (OCE), Amazon Web Services (AWS) and Rightscale.

References

1. Arlitt, M., Jin, T.: A workload characterization study of the 1998 world cup web site. IEEE Network 14(3), 30–37 (2000)
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: Above the clouds: A berkeley view of cloud computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley (February 2009)
3. Bahati, R.M., Bauer, M.A.: Towards adaptive policy-based management. In: NOMS, pp. 511–518 (2010)
4. Bandara, A., Lupu, E., Moffett, J., Russo, A.: A goal-based approach to policy refinement. In: Proceedings of Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, POLICY 2004, pp. 229–239 (2004)
5. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: SOSP 2003: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, pp. 164–177. ACM, New York (2003)
6. Barna, C., Litoiu, M., Ghanbari, H.: Autonomic load-testing framework. In: 2011 International Conference on Autonomic Computing. ACM, New York (2011)
7. Bartolini, C., Stefanelli, C., Tortonesi, M.: Symian: Analysis and performance improvement of the it incident management process. IEEE Transactions on Network and Service Management 7(3), 132–144 (2010)
8. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Comp. Syst. 25(6), 599–616 (2009)
9. Cheng, S.W., Garlan, D.: Stitch: A language for architecture-based self-adaptation (submitted for publication, 2012)
10. Craven, R., Lobo, J., Lupu, E., Russo, A., Sloman, M.: Decomposition techniques for policy refinement. In: 2010 International Conference on Network and Service Management (CNSM), pp. 72–79 (October 2010)
11. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder Policy Specification Language. In: Sloman, M., Lobo, J., Lupu, E.C. (eds.) POLICY 2001. LNCS, vol. 1995, pp. 18–38. Springer, Heidelberg (2001)
12. Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37(10), 46–54 (2004)
13. Ghanbari, H., Simmons, B., Litoiu, M., Iszlai, G.: Exploring alternative approaches to implement an elasticity policy. In: 2011 IEEE International Conference on Cloud Computing (CLOUD), pp. 716–723 (July 2011)
14. Gong, Z., Gu, X., Wilkes, J.: Press: Predictive elastic resource scaling for cloud systems. In: 2010 International Conference on Network and Service Management (CNSM), pp. 9–16 (2010)
15. Hayes, B.: Cloud computing. Commun. ACM 51(7), 9–11 (2008)
16. Kephart, J.O., Walsh, W.E.: An artificial intelligence perspective on autonomic computing policies. In: POLICY 2004: Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks, pp. 3–12. IEEE Computer Society (June 2004)
17. Kephart, J., Chess, D.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)

18. Liaskos, S., Lapouchnian, A., Yu, Y., Yu, E., Mylopoulos, J.: On goal-based variability acquisition and analysis. In: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE 2006). IEEE Computer Society, Minneapolis (2006)
19. Litoiu, M., Woodside, M., Wong, J., Ng, J., Iszlai, G.: A business driven cloud optimization architecture. In: 25th Symposium on Applied Computing. ACM (March 2010)
20. Mell, P., Grance, T.: The nist definition of cloud computing (2009)
21. Moore, B.: Policy core information model (pcim) extensions, rfc 3460 (January 2003)
22. Mylopoulos, J., Chung, L., Liao, S., Wang, H., Yu, E.: Exploring alternatives during requirements analysis. IEEE Software 18(1), 92–96 (2001)
23. Rochwerger, B., Breitgand, D., Levy, E., Galis, A., Nagin, K., Llorente, I.M., Montero, R., Wolfsthal, Y., Elmroth, E., Caceres, J., Ben-Yehuda, M., Emmerich, W., Galan, F.: The reservoir model and architecture for open federated cloud computing. IBM Journal of Research and Development 53(4), 4:1 –4:11 (2009)
24. Rubio-Loyola, J., Charalambides, M., Aib, I., Serrat, J., Pavlou, G., Boutaba, R.: Business-driven management of differentiated services. In: 2010 IEEE Network Operations and Management Symposium (NOMS), pp. 240–247 (2010)
25. Rubio-Loyola, J., Serrat, J., Charalambides, M., Flegkas, P., Pavlou, G.: A methodological approach toward the refinement problem in policy-based management systems. IEEE Communications Magazine 44(10), 60–68 (2006)
26. Schobbens, P.Y., Heymans, P., Trigaux, J.C.: Feature diagrams: A survey and a formal semantics. In: IEEE International Conference on Requirements Engineering, pp. 139–148 (2006)
27. Silva Souza, V.E., Lapouchnian, A., Robinson, W.N., Mylopoulos, J.: Awareness requirements for adaptive systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, pp. 60–69. ACM, New York (2011)
28. Simmons, B.: Strategy-Trees: A Novel Approach To Policy-Based Management. Ph.D. thesis, The University of Western Ontario (2010)
29. Simmons, B., Litoiu, M., Ionescu, D., Iszlai, G.: Towards a cloud optimization architecture using strategy-trees. In: I2TS 2010: Proceedings 9th International Information and Telecommunication Technologies Symposium, Rio de Janeiro, Brazil, December 13-15 (2010)
30. Simmons, B., Lutfiyya, H.: Strategy-Trees: A Feedback Based Approach to Policy Management. In: van der Meer, S., Burgess, M., Denazis, S. (eds.) MACE 2008. LNCS, vol. 5276, pp. 26–37. Springer, Heidelberg (2008)
31. Simmons, B., Lutfiyya, H.: Achieving High-Level Directives Using Strategy-Trees. In: Strassner, J.C., Ghamri-Doudane, Y.M. (eds.) MACE 2009. LNCS, vol. 5844, pp. 44–57. Springer, Heidelberg (2009)

Self-Adaptivity from Different Application Perspectives

Requirements, Realizations, Reflections

Kurt Geihs

University of Kassel, Wilhelmshoeher Allee 73, D-34121 Kassel, Germany
geihs@uni-kassel.de

Abstract. Self-adaptivity can be beneficial in many application domains. In recent years we have researched the engineering of self-adaptive software systems in three rather diverse domains: ubiquitous computing applications, teams of autonomous mobile robots, and management of service-oriented software systems. While all of them perform dynamic adaptation at run-time following a specified control loop, they differ fundamentally in their specific objectives, requirements, properties, and constraints. Consequently, their design and realization focus on different domain aspects and require different modeling and engineering techniques. In this paper we elaborate on synergies and discrepancies in developing the three case studies. We evaluate these self-adaptive systems using a recently published framework for evaluating self-adaptive software systems. The main contributions of this paper are a reflection on the design space of self-adaptive systems and a critique of the proposed evaluation framework.

Keywords: self-adaptive system, run-time adaptation, software engineering, model-driven development, evaluation framework.

1 Introduction

There are many reasons why software systems need to be self-adaptive at run-time. Often self-adaptation occurs as some form of reaction to events that happen in the environment of the running software. Some examples for such events are: a service is no longer available due to service failure; negotiated execution conditions are no longer satisfied; the user situation has changed and requires a different mode of interaction; an application configuration is no longer appropriate because another application suddenly interferes with the application. Other motivations for self-adaptive systems may be to modify the system in order to proactively handle load peaks during certain anticipated situations, or to adapt the execution mode of applications depending on the resource conditions on the computing device. Clearly, reasons for self-adaptation are as manifold as application scenarios and there is a very large spectrum of adaptation requirements and techniques.

Why has self-adaptation moved into the focus of so many research and development activities in the computing realm? Obviously, this major trend is to a large degree due to the widespread availability of powerful and affordable computing devices equipped with different kinds of sensors that provide information on the application

R. de Lemos et al. (Eds.): Self-Adaptive Systems, LNCS 7475, pp. 376–392, 2013.

run-time context and thus enable the applications to react to changes. For example, current smartphones employ up to ten integrated sensors. There are sensors such as brightness indicator, geographical positioning, acceleration, sound, video, compass, gyroscope and object distance. Some of these sensors, e.g. microphone and camera, are built in twice to enhance the usability of the smartphone. This increasing availability of readily accessible sensors in very popular computing devices fosters the development of applications that derive situational knowledge about their environment and that self-adapt in order to provide the best possible service to their users in different situations. In addition, a smartphone provides even more “sensor information” in the form of user-specific information stored in calendars and contact lists that may be taken into account for adaptation reasoning.

Another motivation for adaptivity in software systems is the expectation that adaptive software is easier to maintain and evolve. Hence, it is hoped that built-in adaptivity reduces the cost for the lifecycle management of deployed applications. While context-aware adaptive applications react to changes in their execution environment, adaptation in the sense of software evolution is caused mainly by changing application requirements and infrastructure properties. Consequently, there is a huge interest nowadays in the engineering of self-adaptive systems and applications. Many adaptation mechanisms have been identified and studied. It is generally acknowledged that the development of software with self-adaptive features adds another level of complexity to the software development process. Several software engineering frameworks for self-adaptive software systems have been proposed that help to master the complexity.

In our research group we have explored software adaptivity in three completely different application domains, i.e. mobile ubiquitous computing, teams of autonomous mobile soccer robots, and dynamic service-oriented computing systems. These domains pose very different adaptation requirements and constraints. Nevertheless, from an abstract point of view we were able to employ common adaptation principles and mechanisms in all three cases, leading to substantial synergies and cross fertilization in the design and implementation of software prototypes in these domains. In this paper we discuss lessons learned and open research questions.

In a recent publication Villegas et al. have proposed a comprehensive framework for evaluating quality-driven self-adaptive software systems [21]. It includes classification dimensions, adaptation properties, and a set of metrics to evaluate self-adaptive systems. In this paper we apply this framework to our three case studies. By doing so, we evaluate strengths and weaknesses of the proposed evaluation framework.

The remainder of this paper is organized as follows. First we will introduce the three application domains where we have built adaptive systems: mobile ubiquitous computing (Section 2), teams of autonomous mobile soccer robots (Section 3), and dynamic service-oriented computing systems (Section 4). The presentation of these case studies is kept short and focused on their specific characteristic features. In Section 5 we apply the above mentioned new evaluation framework to our solutions. This also results in a critique of the framework. Finally, Section 6 concludes the paper.

2 Mobile Ubiquitous Computing

The European project MUSIC focused on development support for self-adaptive mobile applications in ubiquitous computing environments [8]. MUSIC has provided a profound model-driven software development methodology complemented by a middleware framework, supporting the implementation and deployment of self-adaptive mobile applications on mobile computing devices [18, 22].

The motivation and requirements for MUSIC were derived from user scenarios where people move around in ubiquitous computing environments and take advantage of computer-mediated assistance. Specifically, such environments are characterized by the following domain assumptions:

- People carry smart networked handheld devices when they are on the move, and use mobile applications that support them in performing everyday tasks in dynamic and different environments.
- Mobile computing environments are highly dynamic: network connections come and go and vary in quality; availability, accessibility and usability of ambient devices change; services available for use appear and disappear and vary in quality; user tasks and needs vary depending on the situation.
- In order to maintain utility and user satisfaction, mobile applications need to adapt their operations according to changes in the environment.
- Mobile applications, similarly to desktop applications, may make use of services available over the Internet to provide enhanced functionalities.
- Mobile applications may interact with external embedded devices and peripherals that are strategically located to both improve functionalities and augment the experience and comfort of the mobile user.
- People switch between multiple mobile applications, which have to be coordinated with respect to their functional and non-functional properties. These applications are usually developed and deployed independently, but may interact.

To demonstrate the wide applicability of the MUSIC approach the project developed a range of different demonstrators that vary in their architectural and functional properties. Additionally, we are building on the MUSIC framework in follow-on research projects where self-adaptive applications are developed by application developers who were not involved in the MUSIC project and had to learn the methodology from scratch.

In this paper we do not present a detailed description of the MUSIC technology, as that has been done elsewhere [10, 12, 18]. Instead, we focus mainly on its adaptation features and methodological development approach.

2.1 Adaptation Mechanisms

MUSIC applications are component-based. The requirements analysis of the MUSIC project revealed the need for the following adaptation mechanisms. All of them are supported in the framework.

- **Compositional adaptation** allows the modification of the application component architecture, i.e. components may be added, removed, or replaced, in order to adapt the functionality of the application. For example, an application may switch from an outdoor GPS navigation component to an indoor navigation service when the user enters a building. In MUSIC, an application component is either atomic or a composite itself, enabling hierarchical decomposition. This hierarchical decomposition creates variation points in the architecture, where there is a choice of alternative component realizations. Thus, by selecting different combinations of component realizations application variants can be derived. In MUSIC we use a developer-defined *utility function* to express how well a given application configuration fits a given context. As properties vary depending on the context, so does the utility of the different application variants. The middleware adapts the running applications to maximize the overall utility.
- **Parametric adaptation** supports fine tuning of applications by changing the values of component parameters. For example, the speaker volume is adjusted in accordance with the environmental noise. Parametric adaptation is an intuitive and effective way to implement variability, but obviously it is less powerful than compositional adaptation [14].
- **Adaptation to external services** extends compositional adaptation by support for dynamic service discovery and binding, such that a local component may be replaced by an externally provided service. For example, when the user enters her car the navigation running locally on her mobile device can be improved by using instead the more powerful and more precise navigation service of the car. In MUSIC, the middleware takes care of service discovery and adaptation reasoning including external services. Service level negotiations and agreements between service client and service provider are supported by the middleware.
- **Deployment adaptation** enables to modify the deployment of the components of a distributed application to the nodes of the computing environment. Component relocation typically occurs when new computing nodes appear or disappear in the environment, or when the quality of the communication path between nodes changes. For example, if a new powerful computing node appears in the dynamic run-time environment, a resource-draining component on a mobile device can be relocated from the mobile to this new node in order to save resources on the mobile device and to speed up the computations. Clearly, there is some overlap with deployment adaptation and adaptation to external services. We will come back to this issue.
- **Device adaptation** supports the adaptation of the hardware resources of the mobile device. For example, it is common practice that users activate GPS only when they need it in an application and switch it off otherwise to save battery. Based on MUSIC, it is possible to automatically adjust the device configuration without user intervention.

While all of these adaptation mechanisms are supported by the MUSIC framework, it is fair to say that compositional adaptation and adaptation to external services were the most interesting research questions, in particular because their modelling concepts are tightly linked to each other and there is no other approach, to the best of our knowledge, that integrates components and services into an adaptation framework for mobile and ubiquitous computing in a comparable way [11]. The list of adaptation mechanisms of MUSIC is not specific for ubiquitous computing applications. Other kinds of distributed applications might also benefit from such mechanisms, provided that the execution environment supports the required reconfigurations. In fact, the two other case studies, presented in the following sections, use some of these mechanisms, but their instantiations differ fundamentally due to specific domain requirements.

2.2 Model-Driven Development

Model-driven software development in MUSIC means that an application developer defines the architectural adaptation model for the application at design time. The application variants and their context dependencies and properties are captured by a *variability model*. Adaptation reasoning is based on a utility function which is part of the variability model of an application. At run-time the utility function is called by the adaptation middleware to evaluate the utility of different application variants and to find the variant with the highest utility in a given situation.

A new UML profile, that we developed using standard UML extension mechanisms, is provided as a modelling language for the variability model. In addition to the application variability and context dependencies, further information is needed to allow the modelling of heterogeneous service and context information in an open ubiquitous computing environment. This information is stored in the domain model, which is an OWL ontology.

In MUSIC, context provision (e.g., context sensors and reasoners) is separated and decoupled from context consumers in order to allow independent development and reuse of context access technologies. Context dependencies can be seen as a link between the context providers and the context consumers and are specified by using the MUSIC Context Query Language (CQL) [16]. Therefore, the modelling notation includes support for the MUSIC CQL as well as for concepts that facilitate the development of context sensors and reasoners.

MUSIC has delivered a software development methodology for self-adaptive applications based on the model-driven architecture (MDA) paradigm together with an integrated tool suite that support platform-independent modelling of the application variability and context dependencies, transformation to code, validation, and testing [20, 22]. The variability and domain models are transformed into code which is executed by the middleware in order to do the adaptation planning. Deliberately, the MUSIC project did not spend any effort on research into software engineering techniques for the development of the application components themselves, because the application components do not play an active role in the adaptation. Component development is considered an orthogonal, well-understood issue [9, 13] and simply outside of the project scope.

2.3 Proof of Concept

The MUSIC approach and framework for self-adaptive applications has been evaluated in many application prototypes during and after the project lifetime. The biggest event was a live demonstration in the Paris Metro in June 2010 showing several adaptive applications, developed by the MUSIC consortium, that aid the Metro traveller in various ways. For example, the Travel Assistant application provides support for the creation of itineraries, station navigation, and retrieval of tourist information. In case of delays it automatically re-calculates the itinerary. Depending on user needs and context, the application offers different levels of functionalities and modes. Notifications can be presented in different modes such as text, speech, or vibration. Navigation information can either be processed on the mobile device or accessed using an external device with a larger screen available in Metro stations. Depending on network availability, the application adapts accordingly. If no network connection is available, all necessary functionality executes locally on the mobile device, allowing the application to function in a stand-alone mode. Otherwise, the application makes use of external 3rd party services, such as the RATP Public Transport Service.

A television report about the demonstration event is available on euronews[1].

3 Teams of Autonomous Mobile Robots

Robot soccer is an exciting and very challenging field for research and experimentation on autonomous mobile robots. The RoboCup tournaments provide an international competition platform for teams of soccer robots [17]. Different kinds of robots play in different RoboCup leagues. Since five years our team has participated successfully in the Middle Size League tournaments of RoboCup. Middle Size League soccer robots are fully autonomous without any remote control, except for referee commands. A team consists of up to five robots. The size of the playing field is 18m x 12m. Robots are custom-built, can move with speeds up to 8 m/s, and communicate via WLAN with their team mates. RoboCup tournaments are great opportunities to test new research results and compare the team performance with other teams.

The cooperative behavior is one of the biggest challenges most pronounced in the RoboCup Middle Size League. The highly dynamic nature of the game requires very swift adaptations by each robot and the team as a whole. In addition, the robots have to cope with unreliable communication and sensory noise that are a matter of fact in the tournament halls, often for unknown reasons. The ability to establish highly reactive teamwork in the presence of unreliable communication and sensory noise is a key to the success of robot soccer teams. Each autonomous robot and the team as a whole are constantly self-adapting to react to the constantly changing game and resource situation.

Let us give an example. If the team executes a plan called Defend, i.e. the opponent team has the ball and is attacking, a popular strategy is called 1-2-1 or diamond

[1] Euronews Futuris,
`http://www.euronews.net/2010/06/16/music-to-you-mobile-s-ears/`

strategy: One robot tries to intercept the opponent dribbler, two robots stay in the mid-field trying to direct the opponent towards a side-line, and another robot moves close to the own goal to secure it if the other three team members fail. The assignment of robots to these tasks is dynamically adapted according to the situation. Should the first robot be outplayed, one of the two mid-field robots will take over its task and rush towards the opponent, while the outplayed robot circles around to assume the free mid-field position. All decisions are made completely autonomously by the individual robots and only later aligned by means of communication.

Clearly, the requirements and constraints of such a self-adaptive technical system are radically different from those of the ubiquitous computing scenario in the previous section. The soccer robot scenario is characterized by the following domain assumptions:

- The most important sensor in a robot is its video camera. Images are processed at a speed of 30 frames per second. The image processing component of a robot tries to detect the important objects on the soccer field, i.e. the ball, the opponents, the white lines, the goal etc. Unfortunately, the video/image processing system is disturbed often by changing light conditions in the tournament hall, by misleading colourful objects in the audience, as well as by other effects. Therefore, video sensor input is inherently unreliable and imprecise, and robots need to collaborate in order to establish a joint view of the world. In addition to the video camera, other types of sensor equipment may be used on a robot, such as laser scanners and infrared sensors, which mainly support the localisation of the robot and the detection of obstacles in its vicinity.
- Robot team members communicate via WLAN. Experience has shown that in tournament situations with large audiences WLAN communication is inherently unreliable. This empirical fact has to be taken into account in the design of the robot cooperation protocols. Therefore, robots cannot rely on and cannot wait for particular messages from their team mates. Instead, each robot must individually make its own decision how to react to an observed game situation and then try to communicate and align its decision with the team mates. The goal is to achieve collaborative behaviour based on individual decisions.
- The ball can move very fast, certainly faster than the robots. Thus, the game situation, i.e. the decision context, changes rapidly all the time. Therefore, adaptation decisions are highly time-critical. The performance of the adaptation management is crucial for the success of the team.
- During a match, soccer robots may break down for various reasons. Or a robot may be sent out by the referee due to a committed rule violation. In such cases, a rapid change in the team strategy is required to cope with such events. Hence, adaptation planning must react flexibly and find the best configuration among many different options.

It is not our intention to present a detailed discussion of the software framework of our soccer robots. Such details can be found in [2]. Instead, we focus mainly on the self-adaptive behavior of the individual robots and the robot team.

3.1 Context Model

Soccer robots, modelled in the software as autonomous agents, need to make swift and autonomous reactive decisions, which cannot always be explicitly communicated beforehand. Taking such decisions bares the risk of degrading the level of coherence within a team. In our approach, this is compensated by the fact that each agent keeps track of its teammates and anticipates their decisions with respect to the common strategy.

Each robot builds and updates continuously its own context model, called *world model.* This model describes the perceived current state of the environment, e.g. the robot's own location, the location of the team mates, the location and direction of the ball, the goals, and more. The world model of a robot is derived from its own sensor input at a frequency of 30 Hz as well as from communicating and merging the world models of the other team members with the local world model on a best effort basis at a frequency of 10 Hz. Discrepancies among the individual world models will happen frequently due to deviations in individual sensor readings and interpretations. Probabilistic filtering techniques are used to align the context models [1].

Due to the highly dynamic nature of the game, the state of the context of a robot is changing continuously and very quickly. Moreover, robots are unable to fully observe the whole state of the game. Hence, they need to estimate the actual state of their environment, and it is very important that they anticipate the dynamic evolution of the situation.

3.2 Behavioral Modeling Language

Obviously, in this application domain reactivity and adaptivity refer to both the activities of the individual robots as well as the collaboration of the complete team. This requires a fundamentally different approach to adaptation modelling compared to the mobile ubiquitous computing applications of the previous section. Key to the specification of cooperative behaviour in our robot team is ALICA, a language for modeling interactive cooperative agents.

In respect to specifying the adaptive behaviour of a robot team, ALICA satisfies the following requirements:

- Single agent behaviours such as tackling, dribbling, and catching the ball need to be combined to multi-agent strategies in a meaningful manner to facilitate the formulation of team actions such as an attack over the side-line, or a diamond defensive formation.
- The tasks in a team strategy need to be independent from the concrete robots executing them, in order to accommodate the dynamic nature of the game and to simplify the development.
- The adaptation should be modelled from a global team perspective from which the behaviour of the single agent is derived.
- The resulting team behaviour should be robust, such that if a malfunctioning robot is detected, the strategy can seamlessly be adapted to work without it.

- The model needs to express how the agents will decide among alternative strategies, if several options are available in a given situation.
- Strategy models should be reusable and not tied to specific situations. This enhances an agent team's adaptability and avoids redundant developer work.

In this paper we will not describe ALICA in detail. The reader is referred to [19] for such a description which also includes the definition of its operational semantics. Here, it is important to note that the language provides a modelling solution for highly dynamic domains in which reactive autonomous agents collaboratively take decisions under tight time constraints and unreliable context information. The team adaptation is as important as the individual adaptations.

3.3 Proof of Concept

We have built a team of soccer robots called CarpeNoctem which consists of six robots and participates successfully in international RoboCup Middle Size League tournaments. In 2009 we achieved a seventh place in the RoboCup world championship in Graz/Austria, in 2010 we won a third place in the international RoboCup GermanOpen in Magedeburg/Germany. The robot team has demonstrated impressively that it is able to quickly adapt its behaviour to the continuously changing game situations.

Clearly, ALICA is one of the cornerstones of these achievements. Thus, we showed experimentally that our adaptation approach satisfies the timing constraints and is robust against a high degree of packet loss and delay. Even with occasional bad network quality, the robots were able to achieve swift adaptations to changing situations, while maintaining team coherence.

4 Dynamic Service-Oriented Architectures

Modern enterprise applications are often built as dynamic service-oriented architectures (SOA) that facilitate dynamic evolution by service rebinding and reconfiguration at run-time. In such environments, a business process is modelled as a service orchestration, i.e. a composition of multiple services that collaboratively realize the business process. The flexibility of a SOA depends on the ability to discover, add, remove, or update services during run-time without stopping the execution of the business process. Dynamic SOA systems are adaptive-systems that monitor the quality of service (QoS) and adapt the service configuration if necessary. QoS contract violations are the primary reason for dynamic reconfiguration.

We are developing a service management framework for dynamic SOA systems. For such kind of adaptive systems we make the following domain assumptions, as far as the adaptive behaviour is concerned:

- A service is characterized by its service type, interface, and QoS attributes. There may be several instances of a service type that differ in their QoS values.

- Service users and service providers conclude service-level agreements (SLA) that formally define the contractually agreed QoS levels.
- QoS monitoring components will detect SLA violations for individual services and for whole service orchestrations, and trigger reactive actions, if needed.
- An adaptation following an SLA violation may involve the replacement of one or more underperforming service instances.
- Service selection is governed by an objective function for the entire orchestration that represents the user expectations and requirements.

For example, let us assume that the response time of some service orchestration is too long, i.e. violates the negotiated threshold. Process monitoring data indicates that a certain sub-process, composed of several services, causes the unexpected delay. The service orchestration is adapted by replacing either the composed sub-process as a whole or by replacing individual service instances such that the overall QoS is restored to acceptable values.

Services may be offered by independent service providers. The matching of service offers and requests in a heterogeneous service landscape raises several challenging questions concerning type compatibility, domain ontologies, semantic annotations, proxy generation, and many more. However, these questions do not play a significant role in the adaptation process. Therefore, we do not discuss how we solved these questions, but refer the reader to the literature [4].

4.1 Monitoring

Service adaptations based on QoS contracts require QoS monitoring at run-time. Our particular focus is on the QoS of composed services, i.e. on the QoS of whole service orchestrations. This requires the aggregation of QoS of (sub-)services. In contrast to the measurement of QoS for single service instances, the aggregated QoS monitoring of business processes needs to take into account the process control structure with constructs such as if-conditions, loops or parallel invocations of services [7]. Thus, the process structure is reflected in the formulas used to compute the aggregated QoS of service orchestrations.

For the QoS monitoring, previous to the process deployment specific QoS sensors are associated to activities in a process. Thus, sensors become part of the process model. Our prototype is built on top of the Oracle BPEL Process Manager [15] and uses the provided sensors to monitor the various activities of the business process. Sensors deliver information such as timestamps when an activity was activated, completed or faulted. The QoS Aggregator computes aggregated QoS values for composed activities and for the whole business process in accordance with the process control structure. Violation of a QoS contract triggers the service selection and adaptation.

4.2 Service Selection and Adaptation

A service orchestration is a composition of multiple services that are required in order to support some business process. We assume that for a given service type there are several instances available that provide the service functionality but have different QoS levels. The QoS of the entire process is computed from the QoS of the services that build up the service composition. Adaptation in this application domain refers to the adaptation of the service orchestration if the overall QoS of the orchestration or the QoS of an individual service instance becomes unsatisfactory. If the QoS monitor reports such an event, the service management will search for a new configuration that satisfies the QoS constraints.

Finding the optimal solution for a given set of service offers means selecting a set of service instances for the orchestration that satisfies the QoS requirements and optimizes a given objective function for the whole orchestration. This objective function depends on the QoS properties of the involved services. It is a kind of utility function that evaluates the utility of different service orchestrations. However, as soon as we have to take several independent QoS dimensions into account, finding the optimal combination from all possible service candidates leads to an NP-hard problem [3]. In [6] we propose several heuristic algorithms for the selection of services in service orchestrations.

The adaptation is controlled and executed by the business process management component which is triggered by the monitoring component. The adaptation reasoning takes the available service offers into account and uses the QoS-based objective function. Executing an adaptation decision involves the rebinding of services to the business process.

4.3 Proof of Concept

The service management framework has been implemented in a prototype based on the Oracle BPEL Process Manager [15]. The functionality and algorithmic performance were tested and evaluated in a variety of simulation experiments, some of them using real services as part of the simulated service environment. These experiments have shown the viability of the conceptual approach and the superior performance of the service selection algorithms [6].

Nevertheless, more experiments in real service-oriented computing environments are needed in order to get more insights into the adaptation planning based on QoS properties as well as into the response times of adaptation management activities. This is future work.

5 Evaluation

The three case studies described in the previous sections address different application domains and consequently were built with different application requirements in mind. In this section we compare the results of the case studies in terms of criteria that were proposed in a recently published comprehensive evaluation framework for self-adaptive systems [21]. While there have been other proposals before for the classification and evaluation of self-adaptive systems, the framework by Villegas et al. stands out because it is based on the analysis of *"over 20 published approaches dealing with* *such systems"* [21]. In this respect it seems to be the most comprehensive and up-to-date investigation. Therefore, we have selected their framework for elaborating on the adaptation characteristics of our three case studies. By doing so, we have discovered some issues with the evaluation framework that deserve further discussions.

5.1 Evaluation Framework

The evaluation framework provides

(i) a set of dimensions for the classification of self-adaptive systems,
(ii) a set of adaptation properties
(iii) a mapping of adaptation properties to quality attributes, and
(iv) a set of quality metrics for quality attributes.

Here, we summarize these issues just briefly, as much as is needed to understand the following discussions. The reader is referred to [21] for the complete story.

(i) Dimensions. There are eight analysis dimensions: adaptation goal *(Why does the system adapt?)*, reference inputs *(How is the adaptation goal represented?)*, measured outputs *(What triggers the adaptation?)*, computed control actions *(How does the adaptation affect the system?)*, system structure *(Can the system structure consisting of adaptation controller and managed system be modified?)*, observable adaptation properties *(What adaptation properties are emphasized?)*, proposed evaluation *(What evaluation criteria are proposed by the system designers?)*, and identified metrics *(What evaluation metrics are proposed by the system designers?)*. The explanatory questions in parentheses are not taken from the paper, but are our own wordings.

(ii) Adaptation properties. These are properties that can be identified and measured in the adaptation process. The framework differentiates between observable adaptation properties in the adaptation controller and in the managed system. The identified observable properties in the controller are stability, accuracy, settling-time, small-overshoot, robustness, termination, consistency, scalability, and security. For the managed system, the identified properties are behavioural/functional invariants, and quality of service conditions.

(iii) Mapping of adaptation properties. A basic assumption of the framework is that the above adaptation properties can be mapped to (and thus measured by) quality attributes such as performance, dependability, safety, and security.

(iv) Quality metrics. These metrics are an attempt to measure quality attributes, and thereby adaptation properties.

While working with the evaluation framework to evaluate our own case studies, we have drawn our own conclusions on the adequacy of these criteria. These will be discussed in-line with the following evaluation.

5.2 Analysis of the Three Case-Studies

The results of the analysis of the three case studies are summarized in Table 1.

Table 1. Applying the evaluation framework dimensions

Case Study	Adaptation Goal	Reference Inputs	Measured Outputs	Control Actions	System Structure	Adaptation Properties	Metrics
MUSIC	situational awareness to improve service to user	degree of user satisfaction with application performance	context data (environment parameters, system parameters, user info)	compositional, parametric, external services, deploym., device	middleware drives adaptation, reconfigurable middleware, hybrid centralized / decentralized adaptation control	stability, scalability, performance, security, small overshoot	planning time, adaptation time
Autonomous, mobile robots	coordination of team action to achieve common goal	depends on currently active plan, activated according to game situation, e.g. plans Defend and Attack have different reference inputs	depends on currently active plan, e.g. distance to ball, distance to goal, relative position of team members etc.	task allocation to team members	decentralized behavior engines (one per robot) compute plans and control actors; evolution of planning through machine learning	short settling time (soft real-time), consistency (of distributed task alloc.), robustness (facing unreliable communications)	task fulfillment (success of coordinated actions)
Dynamic SOA	QoS preservation	QoS contracts (SLA)	monitored QoS of services and service composition	replace and rebind service instances; adapt service composition	service orchestration controlled by central business process mgr.; evolvable QoS management policies	short settling time, consistency, robustness, scalability, security	negotiated QoS parameters

From this table it is obvious that the dimensions provide a rather coarse grained overview of a system's self-adaptation features. Nevertheless, we believe that characterising a self-adaptive system by such adaptation dimensions and properties provides valuable coordinates for positioning a system in the design space of self-adaptive systems which not only is huge but also has a very fuzzy delineation.

5.3 Evaluation Framework Critique

While we found the adaptation dimensions and the adaptation properties rather intuitive and more or less straightforward to apply in order to describe and characterise our case studies, the need for the quality attributes was less obvious. The quality attributes introduced in the evaluation framework, such as performance, dependability and safety, in our opinion are not needed as an intermediate construct in order to map the adaptation properties to specific metrics that allow measuring how much a system satisfies a certain adaptation property. Even in the publication that presents the evaluation framework there are signs that we interpret as supporting indications for our claim. Two examples: In a table showing the mapping of properties to quality attributes, adaptation property “security” is mapped to quality attribute “security”, and “safety” appears both as a quality attribute and as a metric. In summary, we believe that it would be sufficient and appropriate to map the specified adaptation properties directly onto suitable metrics. (If at all possible: for example, it is very difficult and often ambiguous to measure the degree of security of a system.) Therefore, we do not discuss quality attributes of the case studies in this paper.

As a second conclusion from our work with the framework, we found that one aspect is absent, or at least underrepresented, in the evaluation framework that is an important characteristic of self-adaptive systems, i.e. centralized vs. decentralized adaptation control. For example, in our service management framework essentially there is a central business process management component that computes adaptation decisions and triggers reconfiguration activities, while in the robot soccer robot scenario all team members must autonomously decide about their reactions to the evolving game situation. The individual actions are periodically communicated to the team members in order to make the team action consistent. Such a completely decentralized adaptation control is a necessity in this application scenario because of the inherent timing and communication constraints of the environment. It is interesting to note here, that in the adaptation approach of MUSIC we employ basically a centralised approach to application reconfiguration. However, a deployment adaptation requires collaboration of adaptation managers on different nodes of the system. Hence, there is also some aspect of decentralized adaptation control here. A characterization of a self-adaptive distributed system according to “centralized” and “decentralized” adaptation control is an important issue that determines the applicability of a certain adaptation approach for specific application domains.

A minor comment on the comparison with classic control theory: We agree with the authors of the evaluation framework that *“Borrowing properties and metrics from control theory and re-interpreting them for self-adaptive software is not a trivial task.”* It is not at all obvious how the two dimensions Reference Inputs (“the values and corresponding types that are used to specify the state to be achieved and maintained in the managed system by the adaptation mechanism”) and Measured Outputs (“the set of values and corresponding types that are measured in the managed system”) are mapped to self-adaptive applications built on compositional adaptation. For example, what should be called the Reference Input for an adaptive application that reconfigures a multimodal user interface depending on the situation? It is questionable whether the use of classic control theory terminology aids the understanding of self-adaptive software systems.

Last but not least we would like to point to another issue which we were missing in the dimensions of the evaluation framework. Particularly from the perspective of a software engineer one would really like to have information on any methodological support for developing self-adaptive systems based on a certain platform or framework. It is well known that the development of adaptive applications is a complex and challenging task. Software engineering support is needed urgently to master this complexity. However, only very few adaptation approaches come with a coherent development methodology. The MUSIC project has emphasised this requirement and has provided a model-based development methodology for self-adaptive applications

[12]. The application variability model plays a key role in this approach. In general, the provision of an integrated, tool supported development methodology certainly is a distinguishing feature of an adaptation framework and should be represented as a criterion in an evaluation framework for self-adaptive systems.

6 Conclusions

This paper has characterized three kinds of self-adaptive systems in different application domains and has presented an evaluation of these systems based on a recently proposed comprehensive evaluation framework for self-adaptive systems.

The characterization and evaluation underline the breadth and the diversity of the design space for self-adaptive systems. Nevertheless, when designing and building these systems we were able to capture synergies in different realms such as context modelling, context fusion, utility-based objective functions, and service-based adaptation. This is not to say that a single framework, such as MUSIC, would have been sufficient to support the three different application domains. Their specific requirements are just too different: Soccer robots depend on real-time communication and team-oriented adaptation decisions. Adaptation takes place up to 30 times per second which cannot be achieved by the MUSIC framework: (1) due to the exhaustive search for a global optimum of the adaptation decision, while a soccer robot must make a local adaptation decision first, and (2) due to the fact that in MUSIC components may be loaded on demand by the configuration manager while in a soccer robot all components are held in stand-by. Clearly, neither the MUSIC framework nor the SOA platform is capable to satisfy these timing constraints. Dynamic SOA and MUSIC looks like a better match. However, the elaborated support of MUSIC for compositional adaptation incurs development and runtime overhead that is probably not needed in a typical SOA application.

By working with the evaluation framework proposed by Villegas et al. we collected insights in the framework itself that we would like to contribute as feedback to the discussion of such frameworks. We hope that our comments will help to improve the evaluation approach.

One interesting, challenging aspect of evaluating adaptive systems is the question of benchmarks. While this aspect has neither been addressed in our work nor in the cited evaluation framework, Cheng et al. have reported on a benchmarking case study and have contributed general reflections on benchmarks for self-adaptive systems [5]. Looking at the design spectrum of our three case studies, it is absolutely clear that there cannot be a single benchmark for self-adaptive systems. However, it seems like a very attractive research question whether one can develop a kind of generalized benchmark framework that would provide general principles and guidelines for the evaluation of self-adaptive systems and could be tailored to specific application domains such as dynamic SOA systems or ubiquitous computing systems. With the increasing popularity of self-adaptive systems, the demand for such a benchmark framework will increase, too.

Acknowledgements. The author thanks all members of the Distributed Systems Group at the University of Kassel and the European MUSIC team for their contributions to the described projects and the anonymous reviewers for their constructive comments.

References

1. Baer, P., Reichle, R.: Communication and Collaboration in Heterogeneous Teams of Soccer Robots. In: Soccer, R., Lima, P. (eds.). I-Tech Education and Publishing, Wien/Austria (2007) ISBN 978-3-902613-21-9
2. Baer, P., Reichle, R., Geihs, K.: The SPICA Development Framework - Model-Driven Software Development for Autonomous Mobile Robots. In: Intelligent Autonomous Systems 10 (IAS 2010), Baden-Baden, Germany, pp. 211–220 (July 2008)
3. Baligand, F., Rivierre, N., Ledoux, T.: A Declarative Approach for QoS-Aware Web Service Compositions. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 422–428. Springer, Heidelberg (2007)
4. Bleul, S., Zapf, M., Geihs, K.: Flexible Automatic Service Brokering for SOAs. In: 10th IFIP / IEEE Symposium on Integrated Management (IM 2007), Munich, Germany (May 2007)
5. Cheng, S.-W., Garlan, D., Schmerl, B.: Evaluating the effectiveness of the Rainbow self-adaptive system. In: 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Vancouver, BC, Canada (May 2009)
6. Comes, D., Baraki, H., Reichle, R., Zapf, M., Geihs, K.: Heuristic Approaches for QoS-Based Service Selection. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 441–455. Springer, Heidelberg (2010)
7. Comes, D., Bleul, S., Weise, T., Geihs, K.: A Flexible Approach for Business Processes Monitoring. In: Senivongse, T., Oliveira, R. (eds.) DAIS 2009. LNCS, vol. 5523, pp. 116–128. Springer, Heidelberg (2009)
8. EU IST FP6 project MUSIC, `http://www.ist-music.eu`
9. Floch, J., Carrez, C., Cieślak, P., Rój, M., Sanders, R., Shiaa, M.M.: A comprehensive engineering framework for guaranteeing component compatibility. Journal of Systems and Software 83(10), 1759–1779 (2010)
10. Geihs, K., et al.: A Comprehensive Solution for Application-Level Adaptation. Software Practice & Experience 39(4), 385–422 (2009)
11. Geihs, K., Evers, C., Reichle, R., Wagner, M., Khan, M.U.: Development Support for QoS-Aware Service-Adaptation in Ubiquitous Computing Applications. In: Proceedings of DADS Track of ACM SAC 2011, Taichung/Taiwan (2011)
12. 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.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 146–163. Springer, Heidelberg (2009)
13. Kraemer, F.A., Slåtten, V., Herrmann, P.: Tool Support for Rapid Composition, Analysis and Implementation of Reactive Services. Journal of Systems and Software 82(12), 2068–2080 (2009)
14. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing Adaptive Software. IEEE Computer 37(7), 56–64 (2004)
15. Oracle BPEL Process Manager,
`http://www.oracle.com/technology/products/ias/bpel/`

16. Reichle, R., Wagner, M., Khan, M.U., Geihs, K., Valla, M., Fra, C., Paspallis, N., Papadopoulos, G.A.: A Context Query Language for Pervasive Computing Environments. In: Proceedings of IEEE Int. Conf. on Pervasive Computing and Communication, pp. 434–440 (2008)
17. RoboCup Project Homepage, http://www.robocup.org/
18. 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: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 164–182. Springer, Heidelberg (2009)
19. Skubch, H., Wagner, M., Reichle, R., Geihs, K.: A Modelling Language for Cooperative Plans in Highly Dynamic Domains. Mechatronics 2(21), 423–433 (2011)
20. Vanrompay, Y. (ed.): MUSIC Studio and Tools (final version). MUSIC Deliverable D7.3 (2010), http://ist-music.berlios.de
21. Villegas, N., Müller, H., Tamura, G., Duchien, L., Casallas, R.: A Framework for Evaluating Quality-Driven Self-Adaptive Software Systems. In: Proceedings of SEAMS 2011, pp. 80–89 (2011)
22. Wagner, M. (ed.): Modelling notation and software development method for adaptive applications in ubiquitous computing environments (final version). MUSIC Deliverable D6.5 (2010), http://ist-music.berlios.de

Author Index