Javier Cámara   Rogério de Lemos
Carlo Ghezzi   Antónia Lopes (Eds.)

Assurances for Self-Adaptive Systems

Principles, Models, and Techniques

Springer

Volume Editors

Javier Cámara
University of Coimbra, Department of Informatics Engineering
3030-290 Coimbra, Portugal
E-mail: jcmoreno@dei.uc.pt

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

Carlo Ghezzi
Politecnico di Milano, Dipartimento di Elettronica e Informazione
Via Golgi, 42, 20133, Milano, Italy
E-mail: carlo.ghezzi@polimi.it

Antónia Lopes
University of Lisbon, Faculty of Sciences
Campo Grande, 1749-016 Lisbon, Portugal
E-mail: mal@di.fc.ul.pt

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-36248-4 e-ISBN 978-3-642-36249-1
DOI 10.1007/978-3-642-36249-1
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012955856

CR Subject Classification (1998): D.2.1-2, D.2.4, D.2.11, F.3.1-2, D.3.1-2, C.2.4, D.4.5, D.4.7

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

During the past decade, one of the most important challenges in software engineering has been to face the increasing complexity that affects software-intensive systems, regarding not only their development, but most importantly, their operation and maintenance, which cannot be entrusted to human operators because of cost and dependability issues. One of the most successful techniques to date when dealing with these issues is endowing systems with the ability to self-adapt. Such systems monitor themselves at run-time through a variety of probes, reflecting the observed behavior to a control layer that compares it against a model of expected system behavior. When any anomalies or conditions for improvement are detected, they attempt to address the situation (e.g., repair a problem, optimize operation) through a set of effectors placed in the system. Despite recent advances in this area, one key aspect of self-adaptive systems that remains to be tackled in depth is *assurances*: the provision of evidence that the system satisfies its stated functional and non-functional requirements during its operation in the presence of self-adaptation.

This book is one of the outcomes of the ESEC/FSE 2011 Workshop on Assurances for Self-Adaptive Systems (ASAS) held in Szeged, Hungary, in September 2011, which comprised discussions about the fundamental principles, models, methods, techniques, mechanisms, state-of-the-art, and challenges for the provision of assurances in self-adaptive software systems. The book includes extended versions of some of the papers presented during the workshop, as well as invited papers from recognized experts. All the papers in this book were peer-reviewed. The book consists of four parts: "Formal Verification," "Models and Middleware," "Failure Prediction," and "Assurance Techniques."

The first part of the book, entitled "Formal Verification," consists of five papers describing approaches to the formal verification of systems featuring different self-* properties.

The first paper by Cordy, Classen, Heymans, Legay, and Schobbens, entitled "Model Checking Adaptive Software with Featured Transition Systems," presents a formal framework for modeling and analyzing adaptive systems based on featured transition systems, including a model able to capture dynamically changing features in the system and its environment (AFTS), a logic (adaCTL) to express system properties, and algorithms for model checking AFTS models against adaCTL formulae.

The second paper by Filieri and Tamburrelli, entitled "Probabilistic Verification at Runtime for Self-Adaptive Systems," presents an approach to probabilistic verification at run-time for self-adaptive systems based on parametric model checking. Concretely, the authors present a method for the evaluation of the probabilistic logic R-PCTL on parametric discrete-time Markov chains with rewards that relies on algebraic computation.

The third paper by Salaün, Etchevers, De Palma, Boyer, and Coupaye, entitled "Verification of a Self-Configuration Protocol for Distributed Applications in the Cloud," discusses the verification of a configuration protocol for distributed applications in the cloud, where multiple components have to be configured concurrently while respecting some dependencies. The authors use formal verification to check that the protocol for self-configuration complies with a formal specification of its expected behavior, considering aspects such as the order in which components are started, or the correct order of the messages being exchanged.

The fourth paper on formal verification by Nafz, Steghöfer, Seebach, and Reif, entitled "Formal Modeling and Verification of Self-Organizing Systems Based on Observer/Controller-Architectures," presents an approach to formal modeling and compositional verification of self-* systems. To achieve their goal, the authors build upon the use of the observer/controller pattern and a verification approach based on *rely and guarantee*, effectively dividing the verification of the self-* system into two parts: the verification of the functional aspects of the system, and the verification of its self-* features. The approach is illustrated using two different case studies.

The fifth paper by Priesterjahn, Steenken, and Tichy, entitled "Timed Hazard Analysis of Self-Healing Systems," describes an approach for the timed analysis of hazards in component-based self-healing systems. The approach enables the assessment of the effectiveness of reconfiguration operations by determining if these can be completed before the system reaches an unsafe state derived from the propagation of the faults that triggered the reconfiguration in the first place.

Part two of this book, entitled "Models and Middleware," consists of three papers describing approaches on how robustness of autonomous and mobile systems can be improved by employing model-driven development and self-adapting middleware infrastructures.

The first paper by Giese and Schäfer, entitled "Model-Driven Development of Safe Self-Optimizing Mechatronic Systems with MechatronicUML," describes a model-driven development approach that combines modeling using a syntactically and semantically rigorously defined, refined subset of UML and formal verification to deal with safety guarantees in distributed, embedded, real-time systems. Formal verification is based on decomposition and compositional model checking, which enables the scalability of the approach.

The second paper on models, entitled "Model-Based Reasoning for Self-Adaptive Systems — Theory and Practice," by Steinbauer and Wotawa discusses model-based reasoning and its application to self-adaptive systems in the context of autonomous mobile robots. The paper extends the standard sense-plan-act control paradigm with a model-based reasoning engine. The applicability of the proposed approach is demonstrated in the context of a couple of case studies, which involve repairing software at run-time and handling hardware faults in the driving unit of an autonomous mobile robot.

The last paper of this part by Baresi, Guinea, and Saeedi, entitled "Achieving Self-Adaptation Through Dynamic Group Management" discusses a self-adapting middleware infrastructure that exploits the group abstraction to provide designers with powerful means to tackle the design and operation of large, dynamic software systems. The middleware is evaluated in the context of a self-adaptive industrialized greenhouse.

Part three of the book covers "Failure Prediction" and includes two papers on how system reconfiguration can be affected by failure prediction.

In the first paper of this part, entitled "Accurate Proactive Adaptation of Service-Oriented Systems," Metzger, Sammodi, and Pohl review solutions for measuring and ensuring the accuracy of online service quality predictions. They analyze their applicability in the context of third-party services, identify some shortcomings, and propose online testing as an alternative approach to achieve accuracy. The conclusion was that obtaining accurate online quality predictions is still a challenging endeavor.

The second paper "Failure Avoidance in Configurable Systems Through Feature Locality" by Garvin, Cohen, and Dwyer proposes a framework for failure avoidance by reconfiguration in which the framework models individual failure dependence on the system configuration, since these models can be learned more quickly and with less effort. In order to predict the behavior of failures according to historic failure models, the paper exploits a tendency for failures to depend on similar combinations of features. The conclusion is that the adopted technique performs quite well preventing and reconfiguring away from those failures that it targets.

Part four of the book on "Assurance Techniques" contains two papers covering a wide range of techniques.

The first paper of this part, entitled "Emerging Techniques for the Engineering of Self-Adaptive High-Integrity Software," by Calinescu, provides an overview on emerging techniques for the engineering of self-adaptive high-integrity software. It proposes a service-based architecture that aims to integrate these techniques, and discusses opportunities for future research.

The second paper "Assurance of Self-Adaptive Controllers for the Cloud," by Gambi, Toffetti, and Pezzé, discusses the assurance of self-adaptive controllers for the Cloud, and proposes a taxonomy of controllers based on the supported assurance level. The focus of the paper is on the infrastructure as a service (IaaS) layer that takes care of allocating resources to applications. The authors identify two main dimensions for obtaining assurances for self-adaptive controllers, the target levels of assurance and adaptability, and propose a classification of self-adaptive controllers induced by these two dimensions. They also identify combinations of design-time and run-time elements that reach a good compromise between assurance and adaptability, and distinguish some outliers that come from particular choices or uses.

Although the papers in this book have covered a wide range of topics regarding assurances for self-adaptive systems, one could still identify several challenges associated with the field, just to name some: combine development-time rationale with run-time decision making, select and deploy during run-time the appropriate verification and validation tools and techniques for the generation of evidence, and analyze the collected evidence in order to build arguments that should be evaluated against the goals of the system. Moreover, as the system evolves, it may require different degrees of assurance, thus one needs to consider that these assurances need to be dynamically provided depending on the changes that may affect the system, its goals, or the context in which it operates. Nevertheless, we hope that this book will prove valuable for both practitioners and researchers working in the area of assurances for self-adaptive systems, and will be a stepping stone for future research.

We would like to thank all the authors of the book chapters for their excellent contributions, the participants of the ESEC/FSE 2011 Workshop on Assurances for Self-Adaptive Systems (ASAS) for their inspiring participation in moving this field forward, and Alfred Hofmann and his team at Springer for 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: L. Baresi, R. Calinescu, A. Classen, M. Cohen, C.E. da Silva, V. De Florio, N. De Palma, M. Dwyer, H. Giese, L. Grunske, S. Guinea, A. Legay, A. Metzger, R. Mirandola, M. Pezzè, P. Saeedi, G. Salaün, B. Schmerl, P.-Y. Schobbens, H. Seebach, G. Steinbauer, G. Tamburrelli, M. Tichy, G. Toffetti, M. Vieira, F. Wotawa, and several anonymous reviewers.

November 2012

Javier Cámara
Rogério de Lemos
Carlo Ghezzi
Antónia Lopes

Table of Contents

Model Checking Adaptive Software with Featured Transition Systems

Maxime Cordy[1,⋆], Andreas Classen[1], Patrick Heymans[2], Axel Legay[3], and Pierre-Yves Schobbens[1]

[1] PreCISE Research Center, University of Namur, Belgium
{mcr,acs,pys}@info.fundp.ac.be
[2] PreCISE Research Center, University of Namur, Belgium
INRIA Lille-Nord Europe – Universit Lille 1, France
LIFL – CNRS, France
phe@info.fundp.ac.be
[3] INRIA Rennes, France
Aalborg University, Denmark
University of Liège, Belgium
axel.legay@inria.fr

Abstract. We propose to see adaptive systems as systems with highly dynamic features. We model as features both the reconfigurations of the system, but also the changes of the environment, such as failure modes. The resilience of the system can then be defined as the fact that the system can select an adequate reconfiguration for each possible change of the environment. We must take into account that reconfiguration is often a major undertaking for the system: it has a high cost and it might make functions of the system unavailable for some time. These constraints are domain-specific. In this paper, we therefore provide a modelling language to describe these aspects, and a property language to describe the requirements on the adaptive system. We design algorithms that determine how the system must reconfigure itself to satisfy its intended requirements.

1 Introduction

Our society increasingly entrusts computerized systems with complex and critical tasks. These systems have to be adapted, or adapt themselves, to a rapidly evolving environment, while accomplishing their tasks reliably. Due to the short reaction times required, some of these adaptations have to be performed automatically, leading to *self-adaptive* systems. Such systems are usually architected in two levels: The base level manages the basic tasks of the system. It has a simple design that allows rapid response times, but does not allow to respond to exceptional conditions. For instance, a satellite control system is in charge of maintaining the attitude of the satellite so that the solar panels face the sun. It

⋆ FNRS Research Fellow.

must react rapidly when the satellite starts to spin. On the other hand, the adaptive level can detect a change of conditions, and reprogram the base system to adapt to these new conditions. Again, the base system is efficient, but often not exhaustive. For instance, when entering the shadow of a planet, the system will be adapted to give priority to the orientation of the data transmission antenna.

In this paper, we propose to model such adaptation by the notion of *feature*, borrowed from product lines engineering. Classically, a feature is an added functionality to the system, that responds to a (new) need of the customer. Here, a feature can also be an adaption to environmental conditions. For uniformity, we also model such evolutions of the environment (in which we might include some parts of the system itself) as special "features". They can be failures (in which case the associated behaviour describes the failure mode and effects), increase of power of an attacker (in which case the associated behaviour describes the modus operandi of attacks), etc.

In some cases, preserving the functionality of the system is not possible, e.g. in the presence of severe failures. Therefore the requirements need to allow for degraded functionality in such cases, and thus our requirements logic (Section 4) also should include dependency on features. For instance, when a solar panel is damaged, the satellite is allowed to shut down some of its non-prioritary facilities (e.g., observation of aurora borealis) to preserve its vital functions (e.g., avoiding falling on Earth) instead. We thus define *resilience* as the capacity to ensure such conditional requirements in presence of a changing environment (at end of Section 4).

More concretely, our contributions are the following: we propose in Section 3 a fundamental framework, called A-FTS, to model the evolution of both the environment and the adaptive system, and in Section 4 AdaCTL, a temporal logic to describe the requirements on such a system. The next step is, naturally, to check whether the model satisfies the requirements in face of a changing environment, i.e., its *resilience.* This resilience-checking problem departs from the classical model-checking problem in several ways, and thus requires specific adaptations of the classical algorithms, presented in Section 5.

Finally, we compare our approach to extant work in Section 6. This paper only addresses modelling and checking the behaviour of an adaptive system. We briefly sketch the other tools needed for a more comprehensive approach to (self-)adaptive systems in Section 7.

2 Background

In order to make this paper self-contained, we first recapitulate essential definitions related to SPL modelling and verification.

Model checking is a well-known technique for verifying software-intensive systems against temporal properties. In a nutshell, given the model of a system M and a temporal property Φ, a model-checking algorithm determines whether or not M satisfies Φ, written $M \models \Phi$. One may use *labelled transition system* (LTS) as such a model.

Definition 1. *An LTS is a tuple $(S, Act, trans, I, AP, L)$ where S is a set of states, Act is a set of actions, $trans \subseteq S \times S$ is a transition relation, $I \subset S$ is a set of initial states, AP is a set of atomic propositions, and $L : S \rightarrow 2^{AP}$ is a labelling function that associates every states with the set of atomic propositions satisfied by this state.*

We call an execution (or run) of the system an alternating sequence of states and actions. The semantics of an LTS, noted $[\![.]\!]_{LTS}$, is then its set of executions, that is,

$$[\![ts]\!]_{LTS} = \{s_0, \alpha_0, s_1, \alpha_1, \ldots, s_i, \alpha_i, \ldots \mid s_0 \in I \wedge (s_i, \alpha_i, s_{i+1}) \in trans\}. \quad (2.1)$$

In the context of SPLs, the model-checking problem becomes more complex as it requires to identify the exact set of products that do not satisfy a given property [17]. To answer it, one can model each product with an LTS and model-check each of them separately. However, for a SPL of n features, this would require $O(2^n)$ calls to a model checker. Given that distinct products of an SPL may have commonalities in their behaviour, there is a need for concise models and efficient algorithms able to distinguish between commonality and variability.

In this paper, we assume that the variability is captured in a *feature model*, features being atomic units of difference between products. A product is then uniquely defined by a set of features.

Definition 2. *Let F be a set of features. Then a product p is a subset of F, that is, $p \in \mathcal{P}(F)$ where $\mathcal{P}$ denotes the powerset.*

Several representations exist for feature models. Here, we remain at an abstract level and consider feature models independently of their representation. More precisely, we stick to the semantics of Schobbens *et al.* [47] and assume that a feature model defines a set of valid products, *i.e.*, a set of authorized combinations of features.

Definition 3. *A feature model is a couple*

$$d = (F, [\![d]\!] \subseteq \mathcal{P}(\mathcal{P}(F))) \quad (2.2)$$

where F is a set of features, and $[\![d]\!]$ is the set of valid combinations of features.

Given the similarities between different products, SPL modelling approaches aim to capture both their commonality and their variability in a compact manner [8,22,7,17]. Most of them rely on the use of variability operators that determine which parts of the model may or may not be present in a given product; the non-variable parts being shared by all the products. For instance, we proposed *Featured Transition Systems* (FTS) as an extension of Labelled Transition Systems (LTS) meant to model the behaviour of SPLs [17]. FTS model design-time variability of the system behaviour by labelling transitions (i.e., executions of actions) between two states of the system with Boolean constraints defined over the set of features. Then a given product is able to trigger a given transition if and only if its set of features satisfies the associated constraints. Such a constraint is called feature expression and is formally defined as follows.

Definition 4. *A feature expression exp defined over a set of features F is a total function*

$$exp : \mathcal{P}(F) \to \{\top, \bot\}. \tag{2.3}$$

For a given product p, $exp(p)$ returns $\top$ if and only if the features of p satisfies the constraints expressed by exp. In this case, we say that p satisfies exp. We denote by $[\![exp]\!] \subseteq \mathcal{P}(F)$ the set of products that satisfy exp and by $\top$ the feature expression such that $[\![\top]\!] = \mathcal{P}(F)$. In FTS, we use feature expressions to restrict the set of products able to execute a given transition [16].

Definition 5. *An FTS is a tuple $(S, Act, trans, I, AP, L,\ d,\ \gamma)$, where*

- *$S, Act, trans, I, AP, L$ are defined as in Definition 1,*
- *d is a feature diagram,*
- *$\gamma : trans \to \mathcal{P}(F) \to \{\top, \bot\}$ is a total function, labelling each transition with a* feature expression*.*

Thanks to the use of feature expressions, an FTS is a compact representation for a set of LTS, namely one per product. One can obtain the LTS modelling the behaviour of a given product by computing the *projection* of the FTS onto that product [17].

Definition 6. *The projection of an FTS fts to a product $p \in [\![d]\!]$, noted $fts_{|p}$, is the LTS $(S,\ Act,\ trans',\ I,\ AP,\ L)$ where $trans' = \{t \in trans \mid p \in [\![\gamma(t)]\!]\}$.*

The semantics of an FTS $\mathcal{M} = (S, Act, trans, I, AP, L,\ d,\ \gamma)$, noted $[\![fts]\!]_{FTS}$, is then defined as a function that associates a product with the semantics of its projection:

$$[\![\mathcal{M}]\!]_{FTS} : \mathcal{P}(F) \to (S \times Act)^{\omega} : \forall p \in [\![d]\!] \bullet [\![\mathcal{M}]\!]_{FTS}(p) = [\![\mathcal{M}_{|c}]\!]_{LTS} \tag{2.4}$$

Model checking an FTS against a property Φ comes down to distinguishing between the products that satisfy Φ and the ones that do not. This leads us to a new notion of satisfiability $\models_F$, which is not Boolean. Formally, if $\mathcal{M}$ is an FTS defined over a feature model d and Φ is a property, we have

$$(\mathcal{M} \models_F \Phi) = \{p \in [\![d]\!] \bullet \mathcal{M}_{|p} \models \Phi\} \tag{2.5}$$

Given the importance of features in SPLs, a suitable logic should include special operators that reason over them. For this purpose, we defined a featured extension of the *Computational Tree Logic* (CTL) [15]. It is called *fCTL*. Any formula defined in this logic has the form $[\chi]\Phi$ where χ is a feature expression and Φ is a CTL formula. Given an FTS $\mathcal{M}$ defined over a feature model d and a product $p \in [\![d]\!]$, p satisfies $[\chi]\Phi$ if and only if p does not satisfy χ or $\mathcal{M}_{|p}$ satisfies Φ.

3 Modelling the Behaviour of Dynamic SPLs

One way to model a dynamically adaptive software is to represent it as a set of static programs and transitions between them [51]. A transition between two programs models an adaptation of the system, which may be needed in case of changes in the properties of the environment. Since those programs are usually meant to satisfy the same set of high-level goals, they likely share commonalities in their structure and behaviour. Moreover, nowadays an increasing number of software systems are designed as product lines and we observe the emergence of *Dynamic Software Product Lines* (DSPLs) [33], especially in the mobile software industry. In order to cope with changing architectures and environments, these SPLs are equipped with the ability to change their set of features at runtime. In this context, we call *configuration* the set of features of a system at a particular point of time, and *reconfiguration* the process of altering the configuration of this system. The details of such reconfigurations are often hidden to the user who can only witness which *features* of the system have changed.

However, behavioural modelling approaches for SPLs like FTS do not consider that an SPL may have to adapt its behaviour due to unexpected changes in the environment. In other words, a given configuration is chosen and fixed through the whole execution of the system, which is thus completely dependent to the environment. Here, we propose a new modelling formalism that allows dynamic reconfiguration. More precisely, we introduce *Adaptive Featured Transition Systems* (A-FTS), an extension of FTS meant for modelling DSPLs. A-FTS explicitly represent the variability of both the system and its environment. The latter is defined as a set of features, such that an *environment feature* is a Boolean characteristic of the environment that may change over time and that the software has the ability to perceive. The capability of the software to execute a transition thus depends on both its features and those of the environment. For the system, we distinguish between *fixed*, non-mutable features and *adaptable* features. To capture the variability of both the system and the environment, we now define a feature model as a tuple

$$d = (F, F_s \subseteq F, F_a \subseteq F_s, [\![d]\!] \subseteq \mathcal{P}(\mathcal{P}(F))) \tag{3.1}$$

where F is the set of all the features, F_s is the set of the system's features, F_a is the set of the system's adaptable features, and $[\![d]\!]$ the set of valid configurations. We also assume that the system and the environment do not share any feature, so that we can define the set of the environment features (noted F_e) as $F \setminus F_s$. According to the above, we define a system configuration (resp. an environment configuration) as a subset of F_s (resp. F_e), and a (complete) configuration is the union of these two.

An adaptable feature may be enabled or disabled at some points during the execution. Unexpected variations in the environment may force the software to adapt its configuration so that it still works properly according to intended requirements. Given that objective, we consider that after the system executes a transition, it is able to observe the features of the environment that are enabled at that time. According to the current configuration of the environment, the system may alter its own configuration so that it avoids failure or undesirable situations. Since we do not yet consider real-time constraints, we suppose that these reconfigurations are atomic and instantaneous. However, the system can be forbidden to reconfigure itself at some point because it must first terminate the execution of a given sequence of actions. In particular contexts, the environment can also be stable during a given period. The following formal definition of A-FTS takes all these considerations into account.

Definition 7. *An A-FTS is a tuple* $(S, Act, trans, i, AP, L, d, \gamma)$ *where*

- S*,* Act*,* $I \subseteq S$*,* AP*, and* $L : S \to \mathcal{P}(AP)$ *are defined as in Definition 1;*
- $d = (F, F_s, F_a, [\![d]\!])$ *is the feature model modelling the variability of both the system and the environment;*
- $\gamma : S \times Act \times S \to (\mathcal{P}(F) \times \mathcal{P}(F) \to \{\top, \bot\})$ *is a function that defines the transition relation.*

Note that unlike LTS and FTS, the transition relation is not defined as a set called *trans*. Instead, we use the function γ to encode symbolically which transitions exist, which products can execute them and how the configuration of the system and the environment evolve. More precisely, this function allows us to:

1. Determine whether of not the system can reach a state s' by executing an action α from a state s. If that is not the case then $\gamma(s, \alpha, s')$ is a function that returns a $\bot$ whatever the configuration of the system and the environment.
2. Restrict the set of configurations able to execute such a transition. Let us suppose that the the system can reach s' from s by executing α if and only if its features satisfy the feature expression *exp*. For any system's configurations $c \in [\![exp]\!], c' \in \mathcal{P}(F_s)$ and environment's configurations $e, e' \in \mathcal{P}(F_e)$, we have $\gamma(s, \alpha, s')(c \cup e, c' \cup e')$. This definition of transition relation is thus more general than in FTS.
3. Restrain how the configuration of the system and the environment can evolve after the execution of the transition. For instance, let us suppose that if the system moves from s to s' by executing α while in configuration c then it cannot change its configuration at all. To express that constraint we define that for any system's configuration c and environment's configurations e and e', we have for any c' that $\gamma(s, \alpha, s')(c \cup e, c' \cup e') = \bot$.

Note that γ must be defined such that only the adaptable features of the system may be enabled or disabled during runtime:

$$(c \setminus c') \cup (c' \setminus c) \nsubseteq F_a \implies \neg\gamma(s, \alpha, s')(c \cup e, c' \cup e') \tag{3.2}$$

for any $s, \alpha, s, c, c', e, e'$. Moreover, any reconfiguration of the system and the environment must ensure that the new configuration is valid (that is, it must satisfy the constraints of the feature model). Accordingly, the function γ must be such that:

$$c' \cup e' \notin [\![d]\!] \implies \neg\gamma(s, \alpha, s')(c \cup e, c' \cup e') \tag{3.3}$$

for any $s, \alpha, s, c, c', e, e'$.

Example 8. *An example of an A-FTS is graphically illustrated in Figure 1. It depicts a small behavioural model of an adaptive routing protocol inspired by the case study of Zhang et al. [51]. The system has one adaptable feature* encryption*, which leads to two possible configurations. Similarly, the environment has one feature* safe *that may or may not be enabled. Initially, the system is in state* `ready`*. Once it receives a message, it enters the state* `received`*. Then, it routes the message and enters either* `routed-safe` *or* `routed-unsafe` *depending on whether the environment is safe or not. This information is captured by the environment's feature* safe*. In our graphical representation, we model these restrictions by writing the feature expression that must be satisfied by the current configuration of the system and the environment for the transition to be executable. Formally, we make use of the transition relation γ for defining those restrictions. For instance, we know that the transition between* `received` *and* `routed-unsafe` *cannot be executed in a safe environment; accordingly, we define γ such that*

$$\neg\gamma(\texttt{received}, \texttt{route()}, \texttt{routed-unsafe})(c \cup e, c' \cup e') \tag{3.4}$$

where $safe \in e$ and for any e',c,c'. In order to increase readability, we do not explicitly represent the whole definition of the function γ.

If the environment is safe then the system simply sends the message, reaches the state `sent-safe` *and ends up going back to the state* `ready`*. If the environment is not safe and if the system's feature encryption is enabled, then the system encrypts the message and sends it afterwards. Otherwise, it sends the message unencrypted. In every case, the system eventually returns to state* `ready`*. In our graphical representation, the set of atomic propositions satisfied by a given state is written directly below it.*

Additionally, we define that once the system has routed the message it cannot change its configuration until it reaches state `ready` *again. Similarly, we suppose that the feature of the environment are stable between the routing and the sending. We capture those restrictions by means of the transition relation γ. For instance, we have*

$$\gamma(\texttt{routed-safe}, \texttt{send()}, \texttt{sent})(c \cup e, c' \cup e') \Leftrightarrow (c = c' \wedge e = e'). \tag{3.5}$$

for any c, c', e, e'. A property of interest for this system is that while the environment is unsafe, no package is sent before it is encrypted. Further in the paper, we introduce a new logic able to express such properties.

As mentioned in Section 2, an (infinite) execution of a transition system is defined as an alternating sequence of "states" and actions. Unlike LTS and FTS, the concept of state in A-FTS does not only refer to the state of the system itself, but also to its configuration as well as that of the environment. In order to avoid ambiguity, we call that a *macrostate*.

Definition 9. *Let $\mathcal{M}$ be an A-FTS. Then a macrostate of $\mathcal{M}$ is a triplet*

$$(s, c, e) \subseteq S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e).$$

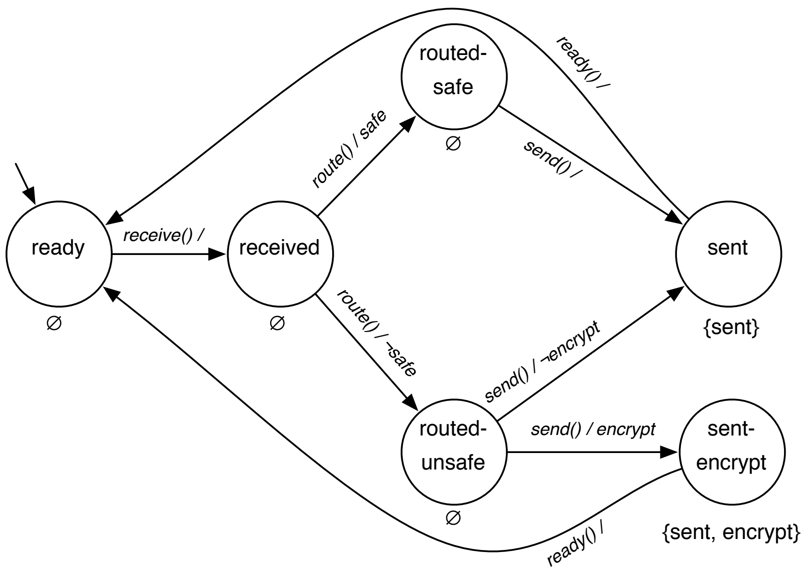


Fig. 1. The A-FTS modelling the adaptive routing protocol

For example, one of the macrostates of the A-FTS presented in Example 8 is $(ready, \emptyset, \{safe\})$. If the A-FTS is in this macrostate, it means that the system is in state *ready*, has not the feature *encryption* enabled and executes in a $safe$ environment. Then once the action `receive()` is executed, the A-FTS can reach one of the following four macrostates: $(\texttt{receive()}, \emptyset, \emptyset)$, $(\texttt{receive()}, \emptyset, safe)$, $(\texttt{receive()}, \textit{encryption}, \emptyset)$, and $(\texttt{receive()}, \textit{encryption}, safe)$. The actual macrostate depends on how the environment evolves and how the system decides to reconfigure itself.

Definition 10. *A run in an A-FTS $\mathcal{M}$ is a sequence of the form*

$$(s_0, c_0, e_0)\alpha_0(s_1, c_1, e_1)\alpha_1 \dots (s_i, c_i, e_i)\alpha_i \dots$$

where $(s_0, c_0, e_0) \in I \times \mathcal{P}(F_s) \times \mathcal{P}(F_e)$ is called the initial *macrostate and where for every $i \in \mathbb{N}$ we have $\gamma(s_i, \alpha_i, s_{i+1})(c_i \cup e_i, c_{i+1} \cup e_{i+1})$.*

Note that this definition allows the system to start in any valid configuration. For instance, a run in the A-FTS described in Example 8 is

$$(ready, \emptyset, \{safe\})\ \texttt{receive()}\ (ready, \{encrypt\}, \emptyset)\ \texttt{route()}\ (routed\text{-}unsafe, \{encrypt\}, \emptyset)\ \texttt{send()}\ (sent\text{-}encrypt)\ \texttt{ready()}\ (ready, \emptyset, \{safe\}) \dots \quad (3.6)$$

As for FTS, we define the projection of an A-FTS onto a configuration c as the A-FTS obtained by setting its initial configuration to c. The resulting A-FTS is noted $\mathcal{M}_{|c}$ and is such that

$$\llbracket \mathcal{M}_{|c} \rrbracket = \{(s_0, c_0, e_0)\alpha_0(s_1, c_1, e_1)\alpha_1 \dots (s_i, c_i, e_i)\alpha_i \dots \in \llbracket \mathcal{M} \rrbracket \mid \\ s_0 \in I \wedge c_0 = c \wedge \forall i \in \mathbb{N} \bullet (s_i, c_i, e_i) \subseteq S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e)\}. \quad (3.7)$$

Then the semantics of an A-FTS is a function that associates a system configuration c with the set of executions where the system starts in configuration c.

Definition 11. *Let $\mathcal{M}$ be an A-FTS. The semantics of $\mathcal{M}$ is the function*

$$\llbracket \mathcal{M} \rrbracket : \mathcal{P}(F_s) \to (S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \times Act)^\omega \bullet \llbracket \mathcal{M} \rrbracket(c) = \llbracket \mathcal{M}_{|c} \rrbracket \quad (3.8)$$

According to the above semantics, there is a close relation between A-FTS, FTS and LTS. An FTS is an A-FTS where the environment has an established, unchanging configuration and where the system starts in any valid configuration and never modifies it, that is

$$\forall (s_0, c_0, e_0)\alpha_0(s_1, c_1, e_1)\alpha_1 \dots (s_i, c_i, e_i)\alpha_i \dots \in \llbracket \mathcal{M} \rrbracket \bullet \\ \forall i \in N \bullet c_i = c_{i+1} \wedge e_i = e_{i+1}. \quad (3.9)$$

A sufficient condition for that condition to hold is that $\gamma(s, \alpha, s')(f, f')$ returns $\perp$ whenever $f \neq f'$. In this case, two runs differ only by (1) the actions chosen by the environment and (2) the states reached by the system. Furthermore, the projection of this FTS onto a configuration c (*i.e.* an LTS) is equivalent to the projection of this special A-FTS to c.

4 The AdaCTL Logic

To express properties that a DSPL must satisfy, we use a variant of the fCTL logic. The resulting logic, called *Adaptive Configuration Time Logic* (AdaCTL), extends the syntax of fCTL to allow further reasoning over the features. Also, its semantics is different because it takes into account that the system and the environment are not always allowed to change their own configuration. In this section, we introduce the syntax and the semantics of AdaCTL and provide an example of properties that can be expressed in this logic.

4.1 Syntax

We can classify the AdaCTL formulae into three categories. The first type of formula is called *feature* formula. It has the form

$$\Psi ::= [\chi]\Phi \quad (4.1)$$

where χ is a feature expression and Φ is a *state* formula. To increase readability, when the feature expression χ is equivalent to $\top$, we omit it; that is, for any state formula Φ, we define that

$$\Phi \triangleq [\top]\Phi. \tag{4.2}$$

A state formula is defined over a set AP of atomic propositions and is built according to the following grammar:

$$\Phi ::= \top \mid a \mid \Psi_1 \wedge \Psi_2 \mid \neg\Psi \mid \mathcal{A}\varphi \mid \mathcal{E}\varphi \tag{4.3}$$

where $a \in AP$, Ψ, Ψ_1 and Ψ_2 are feature formulae, and φ is a *path formula*. This latter category of AdaCTL formula is defined as follows:

$$\varphi ::= \bigcirc\Psi \mid \Psi_1 \mathrm{U}\, \Psi_2 \mid \Psi_1 \mathrm{R}\, \Psi_2 \tag{4.4}$$

where Ψ, Ψ_1, and Ψ_2 are feature formulae, $\bigcirc$ is the *next* operator, U is the *until* operator, and R is the *release* operator.

Before providing AdaCTL with a formal semantics, we first explain it intuitively for each type of formula. A feature formula $[\chi]\Phi$ means that if the system is in a given macrostate (s, c, e) such that the configuration of the system and the environment satisfy the feature expression χ (that is, $c \cup e \in [\![\chi]\!]$), then s must satisfy the state formula Φ. It is thus very similar to an fCTL formula. The difference is that in fCTL, a feature expression occurs before the top-level state formula only, whereas AdaCTL allows it to occur before any state formula. This provides more flexible ways to reason on the features. In particular, if we want to express that a feature f must be enabled, we may use the formula $[\neg f]\ \bot$, which is satisfied if and only if the system is in a configuration where f is enabled. Moreover, while authorizing feature expressions to occur at any level of a formula would not change the expressiveness of fCTL, it increases that of AdaCTL. For example, since the environment is modelled as a set of features varying over time, AdaCTL formulae can model changes of objectives with respect to what happened in the past.

A state formula is a formula defined over a state. Any macrostate satisfies the formula $\top$. A macrostate (s,c,e) satisfies a if and only if a belongs to the set of atomic propositions in $L(s)$; it satisfies the conjunction of two formulae if and only if it satisfies both. Also, the negation of a formula is satisfied if and only if the formula itself is not satisfied. The AdaCTL operator $\mathcal{E}$ is similar to the existential operator of CTL: a macrostate m satisfies the formula $\mathcal{E}\varphi$ if and only if there exists a path starting from m that satisfies φ.

The most subtle difference between AdaCTL and the other two logics lies in the semantics of formulae of the form $\mathcal{A}\varphi$. A macrostate m satisfies $\mathcal{A}\varphi$ if and only if starting from m, the system can ensure by means of reconfigurations that any forthcoming execution will satisfy φ regardless of the environment.

As in CTL, a path π satisfies the AdaCTL path formula $\bigcirc\Psi$ if and only if the first macrostate of π (that is, the macrostate following the initial one) satisfies the feature formula Ψ. A path π satisfies $\Psi_1 \mathrm{U} \Psi_2$ if and only if it eventually reaches a macrostate m_j that satisfies Ψ_2 and every macrostate before m_j on π satisfies Ψ_1. Finally, π satisfies $\Psi_1 \mathrm{R} \Psi_2$ if and only if every macrostate reached by π satisfies Ψ_2 unless a previously reached macrostate satisfied Ψ_1.

From the until and the release operator, one can derive two additional, intensively used operators: *eventually* ($\Diamond$) and *forever* ($\Box$). Intuitively, a path π satisfies $\Diamond\Psi$ if and only if there exists a macrostate along π that satisfies Ψ; π satisfies $\Box\Psi$ if and only if every macrostate along this path satisfies Ψ. Formally, these two operators are obtained as follows:

$$[\chi]\mathcal{E}\Diamond\Psi = [\chi]\mathcal{E}(\top \text{ U } \Psi) \tag{4.5}$$
$$[\chi]\mathcal{A}\Diamond\Psi = [\chi]\mathcal{A}(\top \text{ U } \Psi) \tag{4.6}$$
$$[\chi]\mathcal{E}\Box\Psi = [\chi]\mathcal{E}(\top \text{ R } \Psi) \tag{4.7}$$
$$[\chi]\mathcal{A}\Box\Psi = [\chi]\mathcal{A}(\top \text{ R } \Psi) \tag{4.8}$$

Example 12. *We now provide an example of AdaCTL formula. Let us consider the A-FTS presented in Example 8 and the property according which the system must ensure that in an unsafe environment, no packet is sent before it is encrypted. We can express this property as the AdaCTL formula*

$$\mathcal{A}\Box([\neg safe]\mathcal{A}(\neg sent \text{ U}[\neg safe]encrypted)) \tag{4.9}$$

The $\Box$ operator is needed because the environment can become unsafe at any moment. According to this formula, from every macrostate where the environment feature $safe$ is disabled, the system must eventually reach a macrostate where either the atomic proposition `encrypted` *is satisfied or the environment is safe again; any macrostate reached in the mean time must be such that the atomic proposition* `sent` *is not satisfied.*

Example 13. *Assume we model the requirements for a satellite to normally always maintain altitude (a) and make observations (o), but in case the solar panels are damaged (failure d), the second requirement can be dropped:*

$$\mathcal{A}\Box(\, a \wedge [\neg d]o \,) \tag{4.10}$$

In classical CTL, the R operator is derived from the operator of U and the negation. Consequently, $\Box$ is also derived from $\Diamond$ and the negation. However, as we will show further in this section, this definition is not suitable in *AdaCTL* because $\mathcal{A}$ and $\mathcal{E}$ are not dual. Hence the need for considering R as a primitive operator that cannot be derived from the others.

4.2 Semantics

Before providing AdaCTL with a formal semantics, we first formalise the notions of strategy for both the system and the environment. Intuitively, a strategy for the system determines how the systems reacts (that is, how it reconfigures itself) according to what happened in the past. We call that a *reconfiguration strategy*.

Definition 14. *Let $\mathcal{M} = (S, Act, trans, i, AP, L, d, \gamma)$ be an A-FTS. A* reconfiguration strategy *is a function*

$$Str_C : (S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e))^+ \times S \times \mathcal{P}(F_e) \to \mathcal{P}(F_s) \tag{4.11}$$

where X^* is the type of the sequences of Xs.

Intuitively, the system modifies its configuration according to the sequence of all the macrostates that have been previously visited, the next state that is reached and the next configuration of the environment.

Similarly, we can encode non-deterministic choices and uncontrolled configuration as a strategy for the environment.

Definition 15. *Let $\mathcal{M} = (S, Act, trans, i, AP, L, d, \gamma)$ be an A-FTS. An* environment strategy *is a function*

$$Str_E : (S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e))^+ \to Act \times \mathcal{P}(F_e) \tag{4.12}$$

An environment strategy thus associates the sequence of macrostates that have already been visited with an action and a new configuration for the environment. These definitions of strategy are closely related to those found in the *Alternating Time Logic* (ATL) theory [2]. If the notion of feature were absent, AdaCTL model checking could be regarded as a particular case of ATL model checking. We discuss the link between these two logics more thoroughly in Section 6.

Given an initial macrostate $init = (s_0, c_0, e_0)$, an environment strategy Str_E, and a reconfiguration strategy Str_C, applying Str_E and Str_C from $init$ results in a unique execution

$$Path(init, Str_C, Str_E) = (s_0, c_0, e_0)\alpha_0(s_1, c_1, e_1)\alpha_1 \dots$$

such that $\forall i \in \mathbb{N}$ we have

$$(\alpha_i, e_{i+1}) = Str_E(s_0, c_0, e_0, \dots, s_i, c_i, e_i) \tag{4.13}$$

$$c_{i+1} = Str_C(s_0, c_0, e_0, \dots, s_i, c_i, e_i, s_{i+1}, e_{i+1}). \tag{4.14}$$

This run is *valid* according to $\mathcal{M}$ if and only if it is part of the semantics of $\mathcal{M}$, that is, $Path(Init, Str_C, Str_E) \in [\![\mathcal{M}]\!]$. More generally, we denote by $Path(m, Str_C, Str_E)$ the path starting from a macrostate m induced by the environment strategy $Strat_E$ and the reconfiguration strategy $Strat_C$.

Following the definition of valid execution, we define that an environment strategy Str_E is *valid* according to $\mathcal{M}$ if and only if it cannot lead to invalid executions.

$$\forall init \in I \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \bullet \forall Str_C \bullet Path(init, Str_C, Str_E) \in [\![\mathcal{M}]\!] \tag{4.15}$$

We define similarly the validity of a reconfiguration strategy Str_C:

$$\forall init \in I \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \bullet \forall Str_E \bullet Path(init, Str_C, Str_E) \in [\![\mathcal{M}]\!] \tag{4.16}$$

From now on, we consider only valid strategies. Then, we define the semantics of AdaCTL as follows.

Definition 16. *Let $\mathcal{M}$ be an A-FTS, (s,c,e) one of its macrostates. Then the satisfiability of an AdaCTL feature or state formula by $\mathcal{M}$ in macrostate (s,c,e) is determined according to the following rules:*

$$\begin{array}{rcl}
\mathcal{M},(s,c,e) \models & [\chi]\Phi & \Leftrightarrow c \cup e \notin [\![\chi]\!] \vee \mathcal{M},(s,c,e) \models \Phi \\
\mathcal{M},(s,c,e) \models & \top & \Leftrightarrow \top \\
\mathcal{M},(s,c,e) \models & a & \Leftrightarrow a \in L(s) \\
\mathcal{M},(s,c,e) \models & \Phi_1 \wedge \Phi_2 & \Leftrightarrow \mathcal{M},(s,c,e) \models \Phi_1 \wedge \mathcal{M},(s,c,e) \models \Phi_2 \\
\mathcal{M},(s,c,e) \models & \neg\Phi & \Leftrightarrow \neg(\mathcal{M},(s,c,e) \models \Phi) \\
\mathcal{M},(s,c,e) \models & \mathcal{E}\varphi & \Leftrightarrow \exists Str_C \bullet \exists Str_E \bullet \mathcal{M}, Path((s,c,e), Str_C, Str_E) \models \varphi \\
\mathcal{M},(s,c,e) \models & \mathcal{A}\varphi & \Leftrightarrow \exists Str_C \bullet \forall Str_E \bullet \mathcal{M}, Path((s,c,e), Str_C, Str_E) \models \varphi
\end{array}$$

The semantics of path formulae is very similar to that of CTL path formulae:

$$\begin{array}{rcl}
\mathcal{M},\pi \models & \bigcirc\Psi & \Leftrightarrow \pi[1] \models \Psi \\
\mathcal{M},\pi \models & \Psi_1 \mathrm{U} \Psi_2 & \Leftrightarrow \exists j \geq 0 \bullet \pi[j] \models \Psi_2 \wedge \forall i \leq j \bullet \pi[i] \models \Psi_1 \\
\mathcal{M},\pi \models & \Psi_1 \mathrm{R} \Psi_2 & \Leftrightarrow (\forall j \geq 0 \bullet \pi[j] \models \Psi_2) \vee (\exists i \bullet \pi[i] \models \Psi_1 \wedge \forall k \leq i \bullet \pi[k] \models \Psi_2)
\end{array}$$

where π is a path, $\pi[0]$ is the initial macrostate of π, and $\pi[i+1]$ is the macrostate following $\pi[i]$ in π.

According to the above, we define that $\mathcal{M}_{|c}$ satisfies an AdaCTL formula Ψ if and only if for any initial state i of $\mathcal{M}$ and environment configuration e, $\mathcal{M}$ satisfies the formula in macrostate (i,c,e); that is,

$$(\mathcal{M}_{|c} \models \Psi) \Leftrightarrow \forall i \in I \bullet \forall e \in \mathcal{P}(F_e) \bullet \mathcal{M}(i,c,e) \models \Psi. \tag{4.17}$$

This leads us to the more general satisfiability relation of an AdaCTL formula by an A-FTS. As for FTS and fCTL, this relation, noted $\models_F$ is not boolean [19]. Instead, it is defined as the set of configurations such that when starting in such a configuration, the A-FTS satisfies the formula.

Definition 17. *Let $\mathcal{M}$ be an A-FTS and Ψ an AdaCTL formula. Then,*

$$(\mathcal{M} \models_F \Psi) = \{c \in \mathcal{P}(F_s) \mid \mathcal{M}_{|c} \models \Psi\} \tag{4.18}$$

Definition 18. *A formula Ψ is called an* absolute *requirement if it contains no feature (i.e. no occurrence of the $[\chi]$ operator). It is called* conditional *if it contains non-adaptable system features. It is called* adaptive *if it contains the $\mathcal{A}$ operator. A system is called* adaptive *if it has adaptive requirements and adaptable features.*

Definition 19. *An adaptive system $\mathcal{M}$ with requirements Φ is called* resilient *if there is an initial configuration such that all its requirements are satisfied: $\mathcal{M} \models_F \Phi \neq \emptyset$.*

This implies that, for each adaptive requirement, the system must be equipped with adaptation strategies that allow him to react to any environment (re)configuration, in particular to any failure.

For instance, if $\mathcal{M}$ is the A-FTS presented in Example 8 and Ψ is the adaptive requirement given in Example 12, we have

$$(\mathcal{M} \models_F \Psi) = \mathcal{P}(F_s) \tag{4.19}$$

since for any initial configuration, there exists a reconfiguration strategy that ensures the satisfaction of Ψ. One such strategy would be to enable the feature *encrypt* as soon as the system reaches the state `received`. However, if setting up this feature requires some time (for downloading the encryption code, etc.), the system has to wait that the feature is enabled before sending the message.

Note that according to the above semantics, $\mathcal{A}$ and $\mathcal{E}$ are not dual. We prove this for the $\bigcirc$ operator.

Theorem 20. *Let Ψ be an AdaCTL feature formula. We have*

$$\mathcal{A} \bigcirc \Psi \neq \neg(\mathcal{E} \bigcirc \neg\Psi) \tag{4.20}$$

Proof. Let us assume that $\mathcal{A}$ and $\mathcal{E}$ are dual for the $\bigcirc$ operator. Let us consider the two AdaCTL formulae $\mathcal{A} \bigcirc \mathcal{A} \bigcirc a$ and $\mathcal{E} \bigcirc \mathcal{E} \bigcirc \neg a$ where a is an atomic proposition. Then, for any A-FTS $\mathcal{M}$ and system configuration c, if $\mathcal{M}_{|c}$ satisfies the former then it does not satisfy the latter and vice-versa. Let $\mathcal{M}$ be the A-FTS shown in Figure 2 where f is a feature of the system. It turns out that $\mathcal{M}_{|c} \models \mathcal{A} \bigcirc \mathcal{A} \bigcirc a$ for any configuration c. Indeed, once in state 2 the system can change its configuration such that f is not enabled. In this case, only state 3 is reachable and the formula is thus satisfied regardless of the action chosen by the environment and its configuration. On the other hand, $\mathcal{M}_{|c} \models \mathcal{E} \bigcirc \mathcal{E} \bigcirc \neg a$ for any c as well. Once in state 2, if the system change its configuration such that f is enabled, the system can reach state 4. Since $\mathcal{M}$ satisfies both formulae and given that it is impossible for an A-FTS to satisfy both a formula and its negation, the duality law does not hold. ■

The above counterexample also denies the duality law for the $\Diamond$, the $\Box$, the Uand the Roperators. Since the proof is very similar, we omit it.

Theorem 21. *Let Ψ be an AdaCTL feature formula. We have*

$$\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2) \neq \neg(\mathcal{E}(\neg\Psi_1 \mathrm{R} \neg\Psi_2)) \tag{4.21}$$
$$\mathcal{E}(\Psi_1 \mathrm{U} \Psi_2) \neq \neg(\mathcal{A}(\neg\Psi_1 \mathrm{R} \neg\Psi_2)) \tag{4.22}$$
$$\mathcal{A}\Diamond\Psi \neq \neg(\mathcal{E}\Box\neg\Psi) \tag{4.23}$$
$$\mathcal{E}\Diamond\Psi \neq \neg(\mathcal{A}\Box\neg\Psi). \tag{4.24}$$

This implies that $\mathcal{A}$ cannot be expressed in terms of $\mathcal{E}$. Thus, when model checking an A-FTS against an AdaCTL formula, we have to consider the operator $\mathcal{A}$ and $\mathcal{E}$ separately. On the contrary, the usual *expansion laws* still hold. Basically, an expansion law allows to express an operator in terms of formulae that the

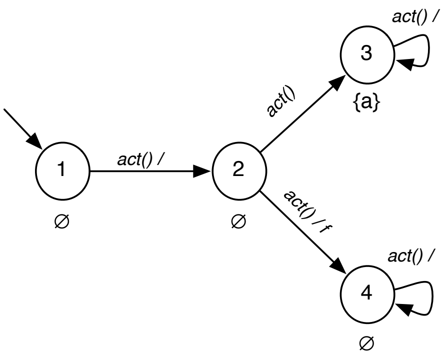


Fig. 2. Counterexample for the duality laws

current state must satisfy to satisfy the formula defined by the operator. For instance, the expansion law for $\mathcal{A}(\Psi_1 \mathrm{U} \Psi 2)$ expresses that this formula is satisfied if and only if Ψ_2 immediately holds or if Ψ_1 holds and the formula holds in the next state. As for CTL, the AdaCTL model checking algorithms make intensively use of these laws, as we will see in the next section.

Theorem 22. *Let Ψ, Ψ_1, Ψ_2 be AdaCTL formulae. Then we have the following expansion laws:*

$$\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2) \equiv \Psi_2 \vee (\Psi_1 \wedge \mathcal{A} \bigcirc \mathcal{A}(\Psi_1 \mathrm{U} \Psi_2)) \tag{4.25}$$
$$\mathcal{A}(\Psi_1 \mathrm{R} \Psi_2) \equiv \Psi_2 \wedge (\Psi_1 \vee \mathcal{A} \bigcirc \mathcal{A}(\Psi_1 \mathrm{R} \Psi_2)) \tag{4.26}$$
$$\mathcal{A}(\Diamond \Psi) \equiv \Psi \wedge \mathcal{A} \bigcirc \mathcal{A} \Diamond \Psi \tag{4.27}$$
$$\mathcal{A}(\Box \Psi) \equiv \Psi \wedge \mathcal{A} \bigcirc \mathcal{A} \Box \Psi \tag{4.28}$$
$$\mathcal{E}(\Psi_1 \mathrm{U} \Psi_2) \equiv \Psi_2 \vee (\Psi_1 \wedge \mathcal{E} \bigcirc \mathcal{E}(\Psi_1 \mathrm{U} \Psi_2)) \tag{4.29}$$
$$\mathcal{E}(\Psi_1 \mathrm{R} \Psi_2) \equiv \Psi_2 \wedge (\Psi_1 \vee \mathcal{E} \bigcirc \mathcal{E}(\Psi_1 \mathrm{R} \Psi_2)) \tag{4.30}$$
$$\mathcal{E}(\Diamond \Psi) \equiv \Psi \wedge \mathcal{E} \bigcirc \mathcal{E} \Diamond \Psi \tag{4.31}$$
$$\mathcal{E}(\Box \Psi) \equiv \Psi \wedge \mathcal{E} \bigcirc \mathcal{E} \Box \Psi \tag{4.32}$$

Proof. We prove only the expansion law of $\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2)$. The other proofs involving the $\mathcal{A}$ operator either can be derived from this one or follow a similar pattern. The proofs related to $\mathcal{E}$ are implied from the expansion laws in CTL.

For any $m = (s, c, e)$, St_E, and St_C, the initial state of $Path(m, St_C, St_E)$ is m and thus depends on neither St_C nor St_E. Accordingly, the semantics of $\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2)$ is equivalent to

$$\begin{aligned}(\mathcal{M}, m \models \Psi_2) \vee ((\mathcal{M}, m \models \Psi_1) \wedge \exists Str_C \bullet \forall Str_E \bullet \\ Path(Path(m, Str_C, Str_E)[1], Str_C, Str_E) \models \Psi_1 \mathrm{U} \Psi_2\end{aligned} \tag{4.33}$$

On the other hand,

$$\mathcal{M}, m \models (\Psi_2 \vee (\Psi_1 \wedge \mathcal{A} \bigcirc \mathcal{A}(\Psi_1 \mathrm{U} \Psi 2))) \tag{4.34}$$

is equivalent to

$$(\mathcal{M}, m \models \Psi_2) \vee ((\mathcal{M}, m \models \Psi_1) \wedge \exists Str'_C \bullet \forall Str'_E \bullet \\ Path(m, Str'_C, Str'_E) \models \bigcirc \mathcal{A}(\Psi_1 \mathrm{U} \Psi_2) \tag{4.35}$$

which can be re-written as

$$(\mathcal{M}, m \models \Psi_2) \vee ((\mathcal{M}, m \models \Psi_1) \wedge \exists Str'_C \bullet \forall Str'_E \bullet \exists Str''_C \bullet \forall Str''_E \bullet \\ Path(Path(m, Str'_C, Str'_E)[1], Str''_C, Str''_E) \models \Psi_1 \mathrm{U} \Psi_2 \tag{4.36}$$

We immediately have that Equation (4.33) implies Equation (4.36); in this case, we have $Str_C = Str'_C = Str''_C$. Next, we define the strategy Str'''_C such that Str'''_C behaves like Str'_C for the first transition, and like Str''_C for the subsequent ones. Then for any Str'''_E we have that

$$Path(Path(m, Str'''_C, Str'''_E)[1], Str'''_C, Str'''_E) \models \Psi_1 \mathrm{U} \Psi_2. \tag{4.37}$$

Equations (4.33) and (4.36) are thus equivalent and we have proven the expansion law. ■

The expansion laws for $\mathcal{A}(\varphi)$ imply that if Str_C is the reconfiguration strategy such that $\forall Str_E \bullet Path(m, Str_C, Str_E) \models \varphi$ then every macrostate m' reached on this path is such that $\mathcal{M}, m' \models \mathcal{A}(\varphi)$.

5 Algorithms

As previously mentioned, model checking an A-FTS against an AdaCTL formula comes down to identifying the initial configurations such that for any initial state and initial environment configuration, the A-FTS satisfies the formula when the system starts in such a configuration. In this section, we propose an algorithm to compute the satisfaction relation between an A-FTS $\mathcal{M}$ and an AdaCTL formula Ψ. For the sake of readability and conciseness, we present only the algorithms in their explicit form. Following our previous work on fCTL model checking [16], we can transform them into symbolic algorithms by using the same encoding techniques. The basic idea is to encode states, configurations, and the transition relation as Boolean functions. Then, symbolic algorithms work with those functions instead of their explicit counterpart (see [16] for more details). Note that the conversion of the transition relation in A-FTS is facilitated since it is already defined as a Boolean function.

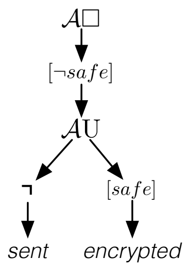


Fig. 3. Parse-tree of the formula $\mathcal{A}(\Box[\neg safe]\mathcal{A}(\neg sent \text{ U } [\neg safe]encrypted))$

5.1 AdaCTL Model Checking

We first decompose Ψ into its *parse tree*. Basically, the parse tree of a formula is a tree such that each node represents a subformula of Ψ, the root is Ψ itself, and the leaves are atomic propositions. For example, the parse tree of the AdaCTL formula given in Example 12 is shown in Figure 3. Then, starting from the leaves and for every subformula Ψ', we compute the set of macrostates

$$Sat(\Psi') = \{(s, c, e) \in S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \mid \mathcal{M}, (s, c, e) \models \Psi'\} \tag{5.1}$$

for every subformula Ψ' of Ψ. Once we have determined the set of macrostates satisfying the whole formula Ψ, we infer the set $(\mathcal{M} \models_F \Psi)$. This method allows us to decompose the verification of a formula into smaller and independent problems. It is thus similar to the fCTL and the standard CTL model checking algorithms [16,15]. However, it has to cope with (1) macrostates instead of states, (2) dynamic features, and (3) the $\mathcal{A}$ quantifier, which does not exist in the other two logics.

The computation of the set of macrostates satisfying feature and state formulae directly follows from their semantics.

$$Sat([\chi]\Phi) = \{(s, c, e) \in S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \mid c \cup e \notin [\![\chi]\!]\} \cup Sat(\Phi) \tag{5.2}$$
$$Sat(\top) = S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \tag{5.3}$$
$$Sat(a) = \{(s, c, e) \in S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \mid a \in L(s)\} \tag{5.4}$$
$$Sat(\Psi_1 \wedge \Psi_2) = Sat(\Psi_1) \cap Sat(\Psi_2) \tag{5.5}$$
$$Sat(\neg\Psi) = S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \setminus Sat(\Psi) \tag{5.6}$$

The first step towards computing a set of the form $Sat(\mathcal{E}\varphi)$ or $Sat(\mathcal{A}\varphi)$ is the definition and the computation of predecessors set. The notion of predecessors is, however, different depending on whether we consider the $\mathcal{E}$ quantifier or the $\mathcal{A}$ quantifier.

In the former case, a macrostate m is an $\mathcal{E}$-predecessor of m' if and only if from m, the macrostate m' can be reached in one transition. More generally, let $\mathcal{S}$ be a set of macrostates. Then the $\mathcal{E}$-predecessors set of $\mathcal{S}$, noted $Pre_{\mathcal{E}}(\mathcal{S})$, is defined as the set of macrostates from which a macrostate in $\mathcal{S}$ can be reached in one transition. Formally,

$$\mathit{Pre}_{\mathcal{E}}(\mathcal{S}) = \{(s,c,e) \mid \exists \alpha \in \mathit{Act}, (s',c',e') \in \mathcal{S} \bullet \\ \gamma(s,\alpha,s')(c \cup e, c' \cup e')\}\}. \quad (5.7)$$

In the latter case, we define that m is an $\mathcal{A}$-predecessor of m' if the system can come up with a valid strategy such that when in m, it is ensured that m' will be reached after the execution of the next transition. Similarly to the previous case, we define the $\mathcal{A}$-predecessor set of a set of macrostates $\mathcal{S}$. Intuitively, it is the set of macrostates such that the system can ensure that after the execution of the next transition, it will reach a macrostate of $\mathcal{S}$. Given that the system chooses its next configuration after the next state and the new environment configuration have been determined, it is defined as

$$\mathit{Pre}_{\mathcal{A}}(\mathcal{S}) = \{(s,c,e) \mid \forall \alpha \in \mathit{Act}, s' \in S, e' \in \mathcal{P}(F_e) \bullet \\ (\exists c'' \in \mathcal{P}(F_s) \bullet \gamma(s,\alpha,s')(c \cup e, c'' \cup e') \Rightarrow \\ (\exists c' \in \mathcal{P}(F_s) \bullet \gamma(s,\alpha,s')(c \cup e, c' \cup e') \wedge (s',c',e') \in \mathcal{S})). \quad (5.8)$$

Let

$$D(s,c,e,\alpha,s',e') = \{(s',c',e') \in S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e) \\ \mid \gamma(s,\alpha,s')(c \cup e, c' \cup e')\} \quad (5.9)$$

then $\mathit{Pre}_{\mathcal{A}}(\mathcal{S})$ is equivalent to

$$\bigcap_{\alpha,s',e'} \{(s,c,e) \mid (D(s,c,e,\alpha,s',e') = \emptyset) \vee (D(s,c,e,\alpha,s',e') \cap \mathcal{S} \neq \emptyset)\} \quad (5.10)$$

Since the semantics of $\mathcal{E}$ is similar to that of the existential quantifier in CTL, the set of macrostates satisfying a formula of the form $\mathcal{E}\varphi$ is computed as in the basic CTL model checking algorithm. $\mathit{Sat}(\mathcal{E} \bigcirc \Psi)$ is the set of macrostate such that in one transition, the system can reach a state s', be in configuration c' and the environment can be in a configuration e' such that (s',c',e') satisfies φ. Formally, we have

$$\mathit{Sat}(\mathcal{E}(\bigcirc\Psi)) = \mathit{Pre}_{\mathcal{E}}(\mathit{Sat}(\Psi)) \quad (5.11)$$

On the other hand, $\mathcal{E}(\Psi_1 \mathrm{U} \Psi_2)$ is the smallest set $\mathcal{S}$ satisfying

$$\mathit{Sat}(\Psi_2) \subseteq \mathcal{S} \quad (5.12)$$

$$\mathit{Sat}(\Psi_1) \cap \mathit{Pre}_{\mathcal{E}}(S) \subseteq \mathcal{S}; \quad (5.13)$$

$\mathcal{E}(\Psi_1 \mathrm{R} \Psi_2)$ is the largest set $\mathcal{S}$ satisfying

$$\mathcal{S} \subseteq \mathit{Sat}(\Psi_2) \quad (5.14)$$

$$\mathcal{S} \subseteq \mathit{Sat}(\Psi_1) \cup \mathit{Pre}_{\mathcal{E}}(\mathcal{S}) \quad (5.15)$$

Both can be computed through fixed-point computation [15]. The corresponding algorithms being identical to their CTL counterpart, we omit them here.

We focus now on the $\mathcal{A}$ quantifier. Basically, the satisfaction sets have a similar definition than those for the $\mathcal{E}$ quantifier; the difference is that the $\mathcal{A}$ quantifier requires the computation of the $\mathcal{A}$-predecessors instead of the $\mathcal{E}$-predecessors. For the next operator, we have the following.

Theorem 23.

$$Sat(\mathcal{A} \bigcirc \Psi) = Pre_{\mathcal{A}}(\Psi) \tag{5.16}$$

Proof. Follows from the semantics of $\mathcal{A} \bigcirc \Psi$ and the definition of $Pre_{\mathcal{A}}(\Psi)$.

We obtain $Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$ through a fixed-point computation. This result relies on the following theorem.

Theorem 24. *$Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$ is the smallest set $\mathcal{S}$ satisfying*

$$Sat(\Psi_2) \subseteq \mathcal{S} \tag{5.17}$$

$$Sat(\Psi_1) \cap Pre_{\mathcal{A}}(\mathcal{S}) \subseteq \mathcal{S} \tag{5.18}$$

Proof. First, it directly follows from the expansion law of the U *operator (see Theorem 22) that $Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$ satisfies Equations (5.17–5.18). It remains to show that any set $\mathcal{S}$ satisfying those Equations is a superset of $Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$.*

Let $m \in Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$. We distinguish between $m \in Sat(\Phi_2)$ and $m \notin Sat(\Phi_2)$.

(a) If $m \in Sat(\Phi_2)$ then $m \in \mathcal{S}$ by Equation 5.17.
(b) Otherwise, by definition of $Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$, we have

$$\exists Str_C \bullet \forall Str_E \bullet Path(m, Str_C, Str_E) \models \Psi_1 \mathrm{U} \Psi_2.$$

It means that Str_C ensures that from m, we eventually reach a macrostate that is in $Sat(\Psi_2)$. Let k be the largest number of transitions needed by Str_C to reach from m such a macrostate, and let m_k this macrostate. Then $m_k \in \mathcal{S}$ by Equation 5.17. Before reaching m_k, we must first reach a macrostate $m_{k-1} \in Sat(\Psi_1)$ such that from m_{k-1}, Str_C ensures to reach m_k in one transition regardless of the strategy of the environment. Such a macrostate is reached in at most $k-1$ transitions, and belongs to $Pre_{\mathcal{A}}(\{m_k\}) \subseteq Pre_{\mathcal{A}}(Sat(\Psi_2))$, and is thus in $\mathcal{S}$ by Equation (5.18). By induction on the maximum number of transitions needed to reach a macrostate in $Sat(\Psi_2)$, we obtain that $m \in \mathcal{S}$.

The proof is then complete. ■

In order to compute $Sat(\Psi_1 \mathrm{U} \Psi_2)$, we define the function

$$T_{\mathrm{U}} : \mathcal{P}(S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e)) \to \mathcal{P}(S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e)) \bullet T_{\mathrm{U}}(\mathcal{S}) = \mathcal{S} \cup (Pre_{\mathcal{A}}(\mathcal{S}) \cap Sat(\Phi_1)). \tag{5.19}$$

Then, according to the Knaster-Tarski theorem and following Theorem 24, this set is the fixed-point of the function T_U when it is first applied to $Sat(\Psi_2)$, that is,

$$Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2)) = \mathcal{S}_i \bullet \forall j \geq i \bullet \mathcal{S}_j = \mathcal{S}_i \tag{5.20}$$

where $\forall j \in \mathbb{N}$

$$\mathcal{S}_0 = Sat(\Psi_2) \tag{5.21}$$

$$\mathcal{S}_{j+1} = T(\mathcal{S}_j) \tag{5.22}$$

The computation of $Sat(\mathcal{A}(\Psi_1 \mathrm{R} \Psi_2))$ follows a very similar procedure. For this reason, we omit the proof.

Theorem 25. *$Sat(\mathcal{A}(\Psi_1 \mathrm{R} \Psi_2))$ is the largest set $\mathcal{S}$ satisfying*

$$\mathcal{S} \subseteq Sat(\Psi_2) \tag{5.23}$$

$$\mathcal{S} \subseteq Sat(\Psi_1) \cup Pre_\mathcal{A}(\mathcal{S}) \tag{5.24}$$

Accordingly, this set can be computed as the fixed-point of the function

$$T_\mathrm{R} : \mathcal{P}(S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e)) \to \mathcal{P}(S \times \mathcal{P}(F_s) \times \mathcal{P}(F_e)) \bullet T_\mathrm{R}(\mathcal{S}) = \mathcal{S} \cap Pre_\mathcal{A}(\mathcal{S}). \tag{5.25}$$

first applied to $Sat(\Psi_2)$.

Example 26. *We illustrate the definitions of $\mathcal{E}$-predecessors and $\mathcal{A}$-predecessors, as well as the above model checking algorithm. Let us consider the small A-FTS shown in Figure 2, which we verify against the formulae $\mathcal{E} \bigcirc \mathcal{E} \bigcirc \neg a$ and $\mathcal{A} \bigcirc \mathcal{A} \bigcirc a$. First, the algorithm determines which macrostates satisfy the atomic proposition a:*

$$Sat(a) = \{(3, c, e)\}.$$

As $\neg a$ occurs in the first formulae, it computes the corresponding satisfaction set by complementing the above:

$$Sat(\neg a) = \{(i, c, e) \mid i \in \{1, 2, 4\}\}.$$

We focus on $\mathcal{E}$-predecessors first. A macrostate satisfies $\mathcal{E} \bigcirc \neg a$ if and only if from this macrostate, the system may reach a set in $Sat(\neg a)$ in one transition. Hence,

$$Sat(\mathcal{E} \bigcirc \neg a) = \{(i, c, e) \mid i \in \{1, 4\} \vee (i = 2 \wedge f \in c)\}.$$

Indeed, from state 1 the system can reach state 2 regardless of its configuration and that of the environment; from state 2 it can reach state 4 if feature f is enabled; and it can always loop on state 4. For the same reasons, the macrostates satisfying the whole property is given by

$$Sat(\mathcal{E} \bigcirc \mathcal{E} \bigcirc \neg a) = \{(i, c, e) \mid i \in \{1, 4\} \vee (i = 2 \wedge f \in c)\}.$$

A macrostate satisfies $\mathcal{A} \bigcirc a$ if and only if from this macrostate, the system can ensure it will reach a macrostate satisfying a. Regardless of the configuration of the system and the environment, the former will remain in state 3 once it reaches this state. Moreover, if the system is in state 2 and feature f is disabled, it will necessarily reach state 3 after the next transition. From state 1, the system cannot reach state 3 in one transition. The corresponding satisfaction set is thus:

$$Sat(\mathcal{A} \bigcirc a) = \{(3, c, e)\} \cup \{(2, c, e) \mid f \notin c\}$$

The definition of $Sat(\mathcal{A} \bigcirc \mathcal{A} \bigcirc a)$ is identical except that this time, the system can satisfy the formula from state 1. The configuration of the system does not matter here since the system may modify it once it reaches state 2. We have

$$Sat(\mathcal{A} \bigcirc \mathcal{A} \bigcirc a) = \{(3, c, e)\} \cup \{(2, c, e) \mid f \notin c\} \cup \{1, c, e\}.$$

5.2 Time Complexity

We now discuss the computational time complexity of checking an A-FTS $\mathcal{M}$ against an arbitrary AdaCTL formula Ψ. Our algorithm recursively computes the satisfactions sets of the subformulae of Ψ. Its time complexity is thus linear in the size of Ψ. The satisfaction sets that are the most costly to compute are those for the U and the R operators. Let us assume that we encode the satisfaction sets and the transition relation symbolically. As in classical CTL model checking, the time complexity of computing $Sat(\mathcal{E}(\Phi_1 \mathrm{U} \Phi_2))$ and $Sat(\mathcal{E}(\Phi_2 \mathrm{R} \Phi_2))$ is bounded by the number of macrostates, i.e., $|S|.2^{|F_s|}.2^{|F_e|}$. This is because when computing the corresponding smallest (resp. greatest) fixed-point, if the fixed-point has not been reached then the application of the corresponding function removes (resp. adds) at least one element. Computing the sets $Sat(\mathcal{A}(\Phi_1 \mathrm{U} \Phi_2))$ and $Sat(\mathcal{A}(\Psi_1 \mathrm{R} \Phi_2))$ is more costly because each application of the functions T_{U} and T_{R} requires the computation of $Pre_{\mathcal{A}}(\mathcal{S})$. The time complexity of this computation is bounded by the number of states multiplied the number of environment configurations, that is, $|S|.2^{|F_e|}$. Since the number of needed applications of T_{U} and T_{R} is also bounded by the number of macrostates, we obtain the following result.

Theorem 27. *The time complexity of model checking an A-FTS against an AdaCTL formula Ψ is bounded by $\mathcal{O}(|S|^2.2^{2.|F_e|}.2^{|F_s|})$.*

Although it is theoretically dominated by $2^{2.|F_e|}$ and is thus in EXPTIME, in practice $|S|^2$ is often bigger.

6 Related Work

To the best of our knowledge, this paper is the first to tackle formal verification of self-adaptive SPLs. Consequently, we can only discuss related work in static SPL model checking and verification of adaptive systems.

6.1 SPL Model Checking

The need for quality assurance techniques in SPLs has been recognized as an important issue and we have observed the emergence of several techniques for solving the SPL model-checking problem. Most of them rely on the use of an automata-based formalism to model the behaviour of an SPL, and on the definition of dedicated checking algorithms. Fischbein *et al.* [30] were the first to propose modal transition systems to model product lines. In a nutshell, modal transition systems are LTS with mandatory and optional transitions, the latter being transitions that are not executable by all the products. To make this formalism more suitable in the context of SPLs, Fantechi and Gnesi [26] enriched it with variability operators and Asirelli *et al.* [6] equipped it with a logic able to express constraints on variable behaviour. Similarly, Gruler *et al.* [36] introduced PL-CCS, an extension of the CCS process algebra with variability operators able to express optionality. Instead of introducing a new formalism, Li *et al.* [40] proposed to model the behaviour of features with independent, single-system models. More precisely, they model both the system without features and the features as finite state machines. Then, the behaviour of a specific product is obtained by clinging its features onto the system.

As explained by Apel *et al.* [4], one can also use single-system model-checking to verify an SPL. In this case, however, the model-checker determines only whether or not there exists a product violating an intended property. The advantage of this approach is that it allows one to benefit from all the existing optimisations implemented in classical model-checkers. However, the goal of our work is more general since we want to pinpoint exactly *all* the misbehaving products or configurations.

The closest work on SPLs verification related to this paper is our previous work about FTS. In [17], we introduced a first definition of FTS, in which transitions are labelled with features and can have priority over each other. We also designed an algorithm for model checking an FTS against an LTL formula. In [16], we provided a new definition of FTS based on feature expressions (*i.e.* the one given in this paper); we also defined the fCTL logic and proposed symbolic model-checking algorithms for FTS. In a recent work [19], we extended the notion of simulation to the context of SPLs and we showed how FTS abstraction can reduce the verification time. In another work, we made use of this new relation to identify special classes of features and reduced the overhead of reverification when such features are introduced in an SPL [18].

The major difference between FTS and A-FTS are (1) the presence of a (possibly hostile) environment, (2) the presence of variability in both the system and its environment, and (3) the ability of the system and the environment to change their features at runtime. Because of those differences, reasoning on A-FTS requires the definition of a new logic to define the properties to be verified, *i.e.*, AdaCTL. In particular, fCTL is not suitable in this context because it permits to reason on neither dynamic features nor an external environment. AdaCTL is thus different from fCTL in both syntax and semantics, in the same way as the classical logics ATL and CTL are different. A comparison between AdaCTL and ATL is given further in this section.

6.2 Verification of Adaptive Systems

The verification of adaptive systems is a topic that received a lot of attention from the scientific community. The work related to that context being particularly large, we focus here on the most recent results.

In their research roadmap for adaptive systems [13], Cheng *et al.* stated that in the context of adaptive systems, the objective of quality assurance is to provide evidence that the system is able to cope with changes in its objectives and its environment. The classical validation and verification methods being meant for stable systems, there is a crucial need for novel techniques specific to adaptive systems. They presented a framework for adaptive systems assurance, in which the system, the goals, and the context are subject to modifications. This results in a succession of models for the system and properties to verify. In our work, such a model is the LTS resulting from fixing the configuration of the state and the environment and removing the transitions unavailable for these configurations; the succession of properties can be expressed by AdaCTL feature formula.

Following the idea of representing adaptive systems as a succession of models, several verification methods model them as a set of programs [1,38,39,51]. To ensure the satisfaction of intended properties in an unstable environment, the system is able to make transitions between those programs. The execution of such transitions is called an adaptation. By performing those, the system modifies its future behaviour. In this context, one distinguishes between local properties that specific programs must satisfy, global properties that must be satisfied by any execution of the system, and transitional properties that must hold during an adaptation.

To specify the transitional properties, Zhang *et al.* proposed a new logic called A-LTL [50]. In their recent work [51], they provided an algorithm based on marking to verify an adaptive system against an A-LTL formula. Although we do not tackle the same problems, there are similarities in their work and ours. Transposed to our work, a program of an adaptive system can be regarded as a particular configuration. Although A-LTL and AdaCTL have incomparable expressiveness, we can also express properties specific to some configurations as well as transitional properties by using AdaCTL feature formulae.

Closer to the notion of dynamic software product lines, Kulkarni *et al.* [39] consider that an adaptive system is a program able to add or remove components during runtime. Our definition of A-FTS is more general, as the effect of features is not limited to the addition or the removal of components. Instead of traditional model checking, they use proof lattice as an alternative solution for verifying that all possible adaptations satisfy all the global properties.

Instead of functional requirements, Filieri *et al.* [27] tackles the verification of non-functional requirements like reliability in the context of adaptive systems. For this purpose, they propose novel algorithms for checking efficiently parametric Markov models (*viz.* parametric discrete-time Markov chains). Combining our work with theirs is an interesting perspective, as it could allow us to quantify the impact of adding or removing features at runtime in terms of reliability, performance or even energy consumption.

6.3 AdaCTL and ATL

As briefly mentioned in Section 4, there is a close relationship between AdaCTL and the *Alternating Time Logic* (ATL) of Alur *et al.* [2]. Without going into much detail or providing formal proofs, we compare our work with theirs and identify the commonalities and differences between the two. ATL is a logic able to express temporal properties on multi-agent systems and concurrent game structures. It provides a special quantifier $<< A >>$ where A is a set of agents or players working together to meet specific goals. The semantics of this operator makes use of a definition of strategy as well : $<< A >> \varphi$ is satisfied if and only if the players in A can find a strategy such that any execution following this strategy satisfies φ. Given that definition, the AdaCTL quantifier $\mathcal{A}$ is clearly similar to $<< Sys >>$ whereas $\mathcal{E}$ is similar to $<< Sys, Env >>$, Sys being the system and Env being the environment.

The difference between the two logics is that in AdaCTL, the transition relation depends on the configuration of both the adaptable and the non-adaptable features. Because of these features, the satisfiability relation is not binary and is thus more general than in ATL. Similarly, an A-FTS with a unique initial configuration can be translated into a two-player turn-based concurrent game structure. In this game, the players are the system and the environment. The action of the former is the choice of its new configuration. For the latter, it is the choice of an action and of its new configuration. In the end, this work can be regarded as a generalization of turn-based, 2-players concurrent game structures in the same way that FTS generalizes LTS.

7 Conclusion

We have presented a well-founded framework for the modelling and analysis of (self-)adaptive systems. We proposed a fundamental model, A-FTS, and a logic, AdaCTL, that are the basis for algorithms for analyzing resilience. This brings a number of benefits:

1. A sound theoretical basis;
2. An integration of static adaptation and dynamic adaptation, in its two variants: external adaptation and self-adaption by applying a pre-programmed change at the adequate point of time;
3. A clear, checkable definition of resilience;
4. Providing counterexamples when resilience fails.

These benefits mainly impact the predictability of self-adaptive systems.

However, we are well aware that many topics need further development before our methodology can be used routinely by engineers. A-FTS are just a fundamental model, that is used by the tools but leads to lengthy descriptions, difficult to manage by humans. It is therefore important to provide more manageable *high-level languages*, that can be compiled (on-the-fly) to A-FTS. These languages need to cover different *levels of abstraction*: the most abstract levels allowing a rapid analysis, while the most detailed can be directly used to generate executable code. This raise the question of *refinement*: can we guarantee that the analysis performed at the abstract level remains valid at the more concrete level? This question can be solved e.g. using alternating refinement [3]. This refinement should be *compositional*, so that modules of the system can be detailed independently, allowing teams to operate in parallel, on one hand, and analysis tools to cope with complexity, on the other.

Some features have a cross-cutting nature: we cannot simply add a module to the system to realize them. That is why we designed A-FTS with a low grain, where individual transitions can refer to features. As a consequence, first, features are spread over the whole A-FTS, and may be difficult to grasp; second, the addition of a supplementary feature is difficult, since each transition might need a revision. We plan thus to use (extensions of) the *aspect-oriented* approach to maintain a more localized and independent description of each feature (previous work on this topic includes [16,31,41]). The addition of a new feature will then hopefully be more understandable. We consider as still unrealistic, though, to hope that all features can be developed without being aware of other features, and that the weaving process will solve all emergent interactions. This raises the problem of defining, detecting and helping to solve *feature interactions*. A number of interesting approaches [34,10] are available, but the problem is still pressing and largely unsolved.

We modelled failures as a subtype of environment features, which allows us to describe failure modes and effects. To have a realistic analysis of failures, we need to also model their *probabilities*, and to integrate (at least) classical methods for failure and reliability analysis [25].

A-FTS is based on the notion of a global state, so that all the information is available to both the system and its environment, and the strategies computed can rely on the unbounded past. In our setting, it is demonstrated that a bounded amount of information about the past is sufficient (finite memory). Techniques such as antichains [28] can be used to heuristically compute strategies that require a small memory. However, assuming complete observation is not realistic in most systems, and in general we need to switch to the notion of *partial observation* [14,5]. Unfortunately, the problem becomes highly complex and often even undecidable [11]. The known (expensive) techniques rely on building all possible evolutions of the environment corresponding to the observations so far [14,37]. A particular, easier problem is the *diagnostic* of failures [46]: from the partial observation provided by its sensors, can the system infer what has failed [46,9]? Can it perform diagnostic actions to help inferring it? Can it remedy to possible failures, even with a partial, ongoing diagnostic [44]? Can it perform the diagnostic against an active adversary [9]?

On the other hand, we have assumed that the self-adaptive system can choose among a set of preprogrammed dynamic system features. This more powerful than it seems, since the system can, if needed, chain several features to construct a complex plan to resist to a hostile environment. In particular, we can model *self-programming* systems by giving as features, the atomic instructions to be assembled. If the atomic instructions are infinite in number, however, our technique does not apply. In the long term, this might become a relevant challenge for deeply self-adaptive autonomous systems.

Since they are domain-specific, we did not consider the *technical means* by which new features will be added to a running system. This usually requires to bring the system to a clean state, where the code can be adapted, downloading the new code, before switching to the new configuration. This reconfiguration might require resources that temporarily decrease the performance of the system. Neither we did consider constraints on the implementation. For instance, a *distributed* implementation might be required [29]. Finally, our current algorithms only provides any strategy satisfying the requirements, but no notion of *preference* among strategies is introduced. In particular, one could prefer strategies that use less memory, require less computation, or minimize some user-defined cost.

Many systems operate in a continuous physical environment. *Hybrid models* can be used to integrate the modelling of the continuous and discrete parts [48,21]. Furthermore, a realistic strategy for a hybrid system must be robust, i.e. able to cope with small disturbances [42]. A first step we did in this direction is to incorporate *real-time* [49,43,20].

In summary, we hope to complement our approach with others, as needed by the vast and multi-disciplinary area of self-adaptive systems, that are expected to progressively enter all domains [32], starting with controllers for space [12], networked systems [35,24], robotics [53], anti-intrusion systems [45], cloud computing [52,23], etc. where the need is most pressing.

References

1. Allen, R., Douence, R., Garlan, D.: Specifying and Analyzing Dynamic Software Architectures. In: Astesiano, E. (ed.) ETAPS 1998 and FASE 1998. LNCS, vol. 1382, pp. 21–37. Springer, Heidelberg (1998)
2. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM 49(5), 672–713 (2002)
3. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y.: Alternating Refinement Relations. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 163–178. Springer, Heidelberg (1998)
4. Apel, S., Speidel, H., Wendler, P., von Rhein, A., Beyer, D.: Feature-interaction detection using feature-aware verification. In: Proceedings of ASE 2011, pp. 372–375. IEEE Computer Society (2011)
5. Arnold, A., Vincent, A., Walukiewicz, I.: Games for synthesis of controllers with partial observation. Theoretical Computer Science 303(1), 7–34 (2003)
6. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: A Logical Framework to Deal with Variability. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 43–58. Springer, Heidelberg (2010)
7. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: Formal description of variability in product families. In: Proceedings of SPLC 2011, pp. 130–139. Springer (2011)

8. Bachmann, F., Goedicke, M., Leite, J., Nord, R.L., Pohl, K., Ramesh, B., Vilbig, A.: A Meta-model for Representing Variability in Product Family Development. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 66–80. Springer, Heidelberg (2004)
9. Bresolin, D., Capiluppi, M.: A game-theoretic approach to fault diagnosis and identification of hybrid systems. Theoretical Computer Science (to appear, 2012)
10. Calder, M., Kolberg, M., Magill, E., Reiff-Marganiec, S.: Feature interaction: a critical review and considered forecast. Computer Networks 41(1), 115–141 (2003)
11. Chatterjee, K., Doyen, L., Henzinger, T.: A survey of partial-observation stochastic parity games. In: Formal Methods in System Design, pp. 1–17 (2012)
12. Chen, W., Saif, M.: Observer-based fault diagnosis of satellite systems subject to time-varying thruster faults. Journal of Dynamic Systems, Measurement, and Control 129(3), 352–356 (2007)
13. 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)
14. Cieslak, R., Desclaux, C., Fawaz, A., Varaiya, P.: Supervisory control of discrete-event processes with partial observations. IEEE Transactions on Automatic Control 33(3), 249–260 (1988)
15. Clarke, E.M., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons using Branching-time Temporal Logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
16. Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A.: Symbolic model checking of software product lines. In: Proceedings of ICSE 2011, pp. 321–330. ACM (2011)
17. 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 ICSE 2010, pp. 335–344. ACM, New York (2010)
18. Cordy, M., Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A.: Managing evolution in software product lines: A model-checking perspective. In: Proceedings of VaMoS 2012, pp. 183–191. ACM (2012)
19. Cordy, M., Classen, A., Perrouin, G., Heymans, P., Schobbens, P.-Y., Legay, A.: Simulation-based abstractions for software product-line model checking. In: Proceedings of ICSE 2012. IEEE (2012)
20. Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A.: Behavioural modelling and verification of real-time software product lines. In: Proceedings of the 16th International Software Product Line Conference, vol. 1, pp. 66–75. ACM (2012)
21. Cury, J., Krogh, B., Niinomi, T.: Synthesis of supervisory controllers for hybrid systems based on approximating automata. IEEE Transactions on Automatic Control 43(4), 564–568 (1998)
22. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005)
23. Di Nitto, E., Ghezzi, C., Metzger, A., Papazoglou, M., Pohl, K.: A journey to highly dynamic, self-adaptive service-based applications. Automated Software Engineering 15(3), 313–341 (2008)

24. 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(2), 223–259 (2006)
25. Ebeling, C.E.: An introduction to reliability and maintainability engineering. McGraw-Hill (1997)
26. Fantechi, A., Gnesi, S.: Formal modeling for product families engineering. In: SPLC, pp. 193–202 (2008)
27. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: Proceedings of ICSE 2011, pp. 341–350 (2011)
28. Filiot, E., Jin, N., Raskin, J.-F.: Antichains and compositional algorithms for ltl synthesis. Formal Methods in System Design 39, 261–296 (2011), doi:10.1007/s10703-011-0115-3
29. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: Proceedings of 20th Annual IEEE Symposium on Logic in Computer Science, LICS 2005, pp. 321–330. IEEE (2005)
30. Fischbein, D., Uchitel, S., Braberman, V.: A foundation for behavioural conformance in software product line architectures. In: Proceedings of ROSATEA 2006, pp. 39–48. ACM Press (2006)
31. Francez, N., Forman, I.R.: Superimposition for Interacting Processes. In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458, pp. 230–245. Springer, Heidelberg (1990)
32. Giese, H., Cheng, B.H.C. (eds.): SEAMS 2011: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. ACM, New York (2011)
33. Gomaa, H., Hussein, M.: Dynamic Software Reconfiguration in Software Product Families. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 435–444. Springer, Heidelberg (2004)
34. Griffeth, N., Lin, Y.-J., et al.: Feature interactions in telecommunications and software systems (FIW, ICFI). IOS Press (1992-2012)
35. Gross, T., Sayama, H.: Adaptive Networks: Theory, Models and Applications. Understanding Complex Systems. Springer (2009)
36. Gruler, A., Leucker, M., Scheidemann, K.: Modeling and Model Checking Software Product Lines. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 113–131. Springer, Heidelberg (2008)
37. Kalyon, G., Le Gall, T., Marchand, H., Massart, T.: Symbolic supervisory control of infinite transition systems under partial observation using abstract interpretation. In: Discrete Event Dynamic Systems, pp. 1–41 (2012)
38. Kramer, J., Magee, J.: Analysing dynamic change in software architectures: A case study. In: Proceedings of the International Conference on Configurable Distributed Systems, Proceedings of CDS 1998, pp. 91–100. IEEE Computer Society, Washington, DC (1998)
39. Kulkarni, S., Biyani, K.: Correctness of Component-Based Adaptation. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 48–58. Springer, Heidelberg (2004)
40. Li, H.C., Krishnamurthi, S., Fisler, K.: Interfaces for modular feature verification. In: Proceedings of ASE 2002, pp. 195–204 (2002)
41. Li, H.C., Krishnamurthi, S., Fisler, K.: Verifying cross-cutting features as open systems. In: SIGSOFT FSE, pp. 89–98 (2002)

42. Majumdar, R., Render, E., Tabuada, P.: Robust discrete synthesis against unspecified disturbances. In: Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control, HSCC 2011, pp. 211–220. ACM, New York (2011)
43. Maler, O., Pnueli, A., Sifakis, J.: On the Synthesis of Discrete Controllers for Timed Systems. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 229–242. Springer, Heidelberg (1995)
44. Paoli, A., Sartini, M., Lafortune, S.: Active fault tolerant control of discrete event systems using online diagnostics. Automatica 47(4), 639–649 (2011)
45. Ragsdale, D., Carver Jr., C., Humphries, J., Pooch, U.: Adaptation techniques for intrusion detection and intrusion response systems. In: 2000 IEEE International Conference on Systems, Man, and Cybernetics, vol. 4, pp. 2344–2349. IEEE (2000)
46. Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., Teneketzis, D.: Diagnosability of discrete event system. IEEE Transactions on Automatic Control 40(9), 1555–1575 (1995)
47. Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., Bontemps, Y.: Feature Diagrams: A Survey and A Formal Semantics. In: Proceedings of RE 2006, pp. 139–148 (2006)
48. Wong-Toi, H.: The synthesis of controllers for linear hybrid automata. In: Proceedings of the 36th IEEE Conference on Decision and Control, vol. 5, pp. 4607–4612. IEEE (1997)
49. Wong-Toi, H., Hoffmann, G.: The control of dense real-time discrete event systems. In: Proceedings of the 30th IEEE Conference on Decision and Control, pp. 1527–1528. IEEE (1991)
50. Zhang, J., Cheng, B.H.: Using temporal logic to specify adaptive program semantics. Journal of Systems and Software 79(10), 1361–1369 (2006)
51. Zhang, J., Goldsby, H.J., Cheng, B.H.: Modular verification of dynamically adaptive systems. In: Proceedings of AOSD 2009, pp. 161–172. ACM, New York (2009)
52. Zhang, Q., Cheng, L., Boutaba, R.: Cloud computing: state-of-the-art and research challenges. Journal of Internet Services and Applications 1(1), 7–18 (2010)
53. Zhong, C., DeLoach, S.A.: Runtime models for automatic reorganization of multi-robot systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, pp. 20–29. ACM, New York (2011)

Probabilistic Verification at Runtime for Self-Adaptive Systems

Antonio Filieri and Giordano Tamburrelli

DeepSE Group @ DEI - Politecnico di Milano, Italy
{filieri,tamburrelli}@elet.polimi.it

Abstract. An effective design of effective and efficient self-adaptive systems may rely on several existing approaches. Software models and model checking techniques at run time represent one of them since they support automatic reasoning about such changes, detect harmful configurations, and potentially enable appropriate (self-)reactions. However, traditional model checking techniques and tools may not be applied as they are at run time, since they hardly meet the constraints imposed by on-the-fly analysis, in terms of execution time and memory occupation. For this reason, efficient run-time model checking represents a crucial research challenge.

This paper precisely addresses this issue and focuses on probabilistic run-time model checking in which reliability models are given in terms of Discrete Time Markov Chains which are verified at run-time against a set of requirements expressed as logical formulae. In particular, the paper discusses the use of probabilistic model checking at run-time for self-adaptive systems by surveying and comparing the existing approaches divided in two categories: state-elimination algorithms and algebra-based algorithms. The discussion is supported by a realistic example and by empirical experiments.

1 Introduction

Software is the driving engine of modern society. Most human activities, including the most critical ones, are either software enabled or entirely managed by software. As software is becoming ubiquitous and society increasingly relies on it, the adverse impact of unreliable or unpredictable software cannot be tolerated. Indeed, software systems have to be able to evolve correspondingly to their deployment environment in order to guarantee a seamless fulfillment of desired requirements and ensure a minimal downtime. In response to this challenge, current Software Engineering aims at designing *Self-Adaptive Systems* which are able to react and reconfigure themselves minimizing human intervention and ideally guaranteeing a lifelong requirement fulfillment. To date, Software Engineering research in self-adaptive systems has produced promising initial results, as illustrated for example in [24]. However, even if these findings provide an essential step towards a set of effective and efficient solutions for self-adaptation, they are not the end of the story as building these dependable systems is still unclear and requires further investigation.

Designers must ensure that any critical requirement of the system continues to be satisfied before, during and after unforeseen scenarios. By this we mean that software systems are required to be *dependable*, to avoid damaging effects due to violated requirements that can range from loss of business to loss of human lives. At the same time, the complexity of modern software systems has grown enormously in the past years with users always demanding for new features and better quality of service. Software systems changed from being monolithic and centralized to modular, distributed, and dynamic. They are increasingly composed of heterogeneous components and infrastructures on which software is configured and deployed. When an application is initially designed, software engineers often only have a partial and incomplete knowledge of the external environment in which the application will be embedded at run time. Design may therefore be subject to high uncertainty. This is further exacerbated by the fact that the structure of the application, in terms of components and interconnections, often changes dynamically. New components may become available and published by providers for use by potential clients. Some components may disappear, or become obsolete, and new ones may be discovered dynamically. This may happen, for example, in the case of Web service-based systems [6,7]. This also happens in pervasive computing scenarios where devices that run application components are mobile [27]. Because of mobility, and more generally context change, certain components may become unreachable, while others become visible during the application's lifetime. Finally, requirements also change continuously and unpredictably, in a way that is hard to anticipate when systems are initially built. Because of uncertainty and continuous external changes the software application is subject to continuous adaptation and evolution.

This paper focuses on how analyzing and comparing existing approaches aimed at managing run-time changes by verifying that the software evolves dynamically without disrupting the dependability of applications. Dependability includes attributes such as reliability, availability, performance, safety, security. In this paper we focus our attention on two main dependability requirements that typically arise in the case of decentralized and distributed applications: namely, *reliability* and *performance*. Both reliability and performance depend on environment conditions that are hard to predict at design time, and are subject to a high degree of uncertainty. For example, performance may depend on end-user profiles, on network congestion, on load conditions of external services that are integrated in the application. Similarly, reliability may depend on the behavior of the network and of the external services that compose the application being built.

Existing approaches focus on supporting the development and operation of complex and dynamically evolvable software systems leveraging on *Formal Models*. These formal models are built at design time to support an initial assessment that the application satisfies the requirements. Models are then kept alive at run time and continuously verified to check that the changes with respect to the design-time assumptions do not lead to requirements violations. This requires efficient mechanisms for run-time verification. If requirements violations are detected, appropriate actions must be undertaken, ranging from off-line evolution to on-line adaptation. In particular, much research is currently investigating the extent to which the software can respond to predicted or detected requirements violation through self-managed reactions, in an autonomic manner. These, however, are out of the scope of this paper, which only focuses on describing run-time verification approaches.

Verification at runtime of reliability and performance properties for self-adaptive systems typically relies on *Probabilistic Models* such as: *Discrete Time Markov Chains* and *Discrete Time Markov Rewards Models*. This paper introduces these formalisms and subsequently illustrates and compares the approaches for efficient runtime verification. In particular, our contribution is structured as follows. Section 2 describe the mathematical foundations of Markov Chains and PCTL (i.e., the logic commonly adopted to verify properties on discrete Markov models). Such mathematical concepts are described by relying on a realistic example also introduced in this section and used throughout the paper to illustrate the different model checking techniques. Section 3 dicusses and compares the existing approaches for run-time verification. Finally, 5 draws some remarking conclusions and illustrates potential future work.

2 Probabilistic Models for Run-Time Verification

This section provides an introduction to the probabilistic models adopted to express reliability and performance properties for self-adaptive systems. In this section we provide a brief introduction to the mathematical concepts used throughout the paper. In particular in Sections 2.3 and 2.4 we describe respectively the *Probabilistic Computational Tree Logic* and its extension with rewards used to represent properties of systems to be verified at runtime. The reader can refer to [4,5] for a comprehensive in-depth treatment of these concepts.

2.1 Discrete Time Markov Chains

Discrete Time Markov Chains (DTMCs) are a widely accepted formalism to model reliability of systems built by integrating different components. In particular, they proved to be useful for an early assessment or prediction of reliability [21]. The adoption of DTMCs implies that the modeled system meets, with some tolerable approximation, the Markov property–described below. This issue can be easily verified as discussed in [8,14].

DTMCs are discrete stochastic processes with the Markov property, according to which the probability distribution of future states depends only upon the current state. They are defined as a Kripke structure with probabilistic transitions among states. *States* represent possible configurations of the system. *Transitions* among states occur at discrete time and have an associated probability.

Formally, a (labeled) DTMC is a tuple $(S, S_0, \mathbf{P}, L)$ where

- S is a finite set of states
- $S_0 \subseteq S$ is a set of initial states
- $\mathbf{P} : S \times S \to [0, 1]$ is a stochastic matrix $(\forall s \in S \mid \sum_{s' \in S} \mathbf{P}(s, s') = 1)$. An element $\mathbf{P}(s_i, s_j)$ represents the probability that the next state of the process will be s_j given that the current state is s_i.
- $L : S \to 2^{AP}$ is a labeling function. AP is a set of atomic propositions. The labeling function associates to each state the set of atomic propositions that are true in that state.

A state $s \in S$ is said to be an *absorbing state* if $\mathbf{P}(s, s) = 1$ otherwise the state is a *transient state*. If a DTMC contains at least one absorbing state, the DTMC itself is said to be an *absorbing DTMC*. We further assume that in our models for any transient state there is a non zero probability of reaching at least one of the absorbing states. In the simplest model for reliability analysis, the DTMC will have two absorbing states, representing the correct accomplishment of the task and the task's failure, respectively. The use of absorbing states is commonly extended to modeling different failure conditions. For example, different failure states may be associated with the invocation of different external services. The set of failures to look for is strictly domain-dependent.

In an absorbing DTMC with r absorbing states and t transient states, rows and columns of the transition matrix $\mathbf{P}$ can be reordered such that $\mathbf{P}$ is in the following *canonical form*:

$$\mathbf{P} = \begin{pmatrix} Q & R \\ 0 & I \end{pmatrix} \tag{1}$$

where I is an r by r identity matrix, 0 is an r by t zero matrix, R is a nonzero t by r matrix and Q is a t by t matrix.

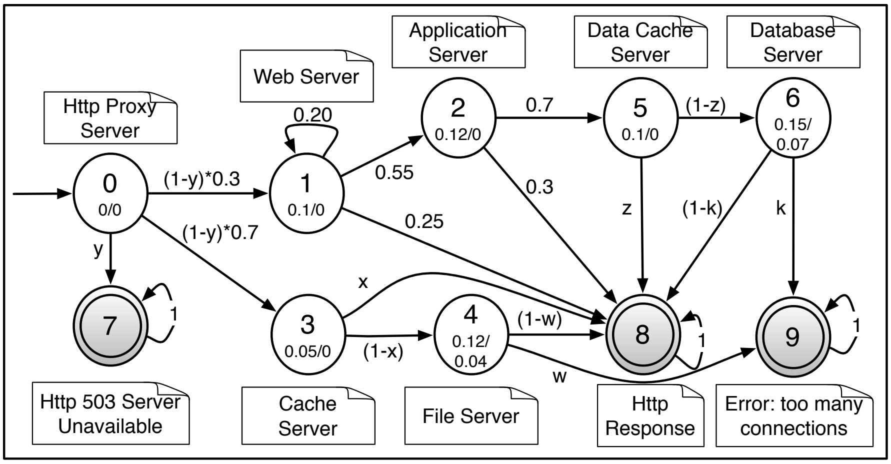


Fig. 1. DTMC Example

Consider now two distinct transient states s_i and s_j. The probability of moving from s_i to s_j in exactly 2 steps is $\sum_{s_x \in S} P(s_i, s_x) \cdot P(s_x, s_j)$. Generalizing, for a k-steps path and recalling the definition of matrix product, it follows that the probability of moving from any transient state s_i to any other transient state s_j in exactly k steps corresponds to the entry (s_i, s_j) of the matrix Q^k. As a natural generalization, we can define Q^0, which represents the probability of moving from each state s_i to s_j in 0 steps, as the identity t by t matrix, whose elements are 1 iff $s_i = s_j$ [15].

Due to the fact that R must be a nonzero matrix, and $\mathbf{P}$ is a stochastic matrix, Q has uniform-norm strictly less than 1, thus $Q^n \to 0$ as $n \to \infty$, which implies that eventually the process will be absorbed with probability 1.

Let us consider for example the DTMC in Figure 1, which represents a typical web architecture. The system comprises an HTTP Proxy server, a Web server and an application server. In addition, structured data and static content (e.g., files, images, etc.) are respectively stored in a Database server and File server. Both of them are cached by ad-hoc cache servers. Each state is labelled by a numeric label and by a couple in the form n_1/n_2, its meaning will be clear later on. States 7, 8 and 9 are absorbing states. The former represents the failure of serving an incoming request due to an unavailable server (e.g., overloaded server or maintenance downtime). The latter represent the endpoint of a correct HTTP request. Transitions among non-absorbing states reports the probability for an HTTP request of passing from one element of the architecture to the other. For example transition $0-1$ indicates the probability that a request is associated to static or dynamic content. Transition $1-1$ indicates instead the probability of an HTTP self-redirect.

Conversely, transitions to absorbing states indicates the final outcome in processing a request. We use variables as transition labels to indicate that the value of the corresponding probability is unknown, and may change over time. For example transitions $3-4$ and $5-6$ indicate the cache hit probability.

In matrix form, the same DTMC would be characterized by the following matrices Q and R:

$$Q = \begin{pmatrix} 0 & (1-y)0.3 & 0 & (1-y)0.7 & 0 & 0 & 0 \\ 0 & 0.2 & 0.55 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.7 & 0 \\ 0 & 0 & 0 & 0 & 1-x & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1-z \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} y & 0 & 0 \\ 0 & 0.25 & 0 \\ 0 & 0.3 & 0 \\ 0 & x & 0 \\ 0 & 1-w & w \\ 0 & z & 0 \\ 0 & 1-k & k \end{pmatrix}$$

Notice that the parameters necessary to model a system with a DTMC (i.e., the values in the DTMC matrix) may be obtained by experimental results as well as by estimates extracted by similar systems or by previous version of the system under design. Finally, the system reported in this section is just a toy example that we use to introduce the proposed approach. However, the concepts described hereafter apply seamlessly to real systems which might have thousands of states and failures.

2.2 Discrete Time Markov Rewards Models

A D-MRM [1] is a DTMC augmented with rewards, through which one can quantify a benefit (or loss) due to the residence in a specific state or the move along a certain transition. A D-MRM has an underlying DTMC, through which designers can provide a high-level model for the system's control flow by abstracting the execution state space into a finite set of abstract states relevant to the verification[1]. As illustrated later on, D-MRM may be used to model performance properties or even properties concerning costs (e.g., energy consumptions).

A reward is a non-negative value assigned to a state or a transition. Rewards can represent information such as average execution time, power consumption, number of I/O operations, or even cost of an outsourced operation. A D-MRM is a tuple (S, S_0, P, L, ρ, i) where:

- S is a finite set of states,
- $S_0 \subseteq S$ is a set of initial states,
- $P : S \times S \to [0,1]$ is a stochastic matrix $(\forall s \in S \mid \sum_{s' \in S} P(s, s') = 1)$. An element $P(s_i, s_j)$ represents the probability that the next state of the process will be s_j given that the current state is s_i,
- $L : S \to 2^{AP}$ is a labeling function which assigns to each state the set of *Atomic Propositions* that are true in the state,
- $\rho : S \to \mathbb{R}_{\geq 0}$ is a *state reward* function assigning to each state a non-negative real number,
- $\iota : S \times S \to \mathbb{R}_{\geq 0}$ is a *transition reward* function assigning a non-negative real number to each transition.

To understand how rewards are gained, we need to precisely state how the system modeled by the D-MRM evolves over a sequence of time steps. At step 0 the system enters the initial state s_0. At step 1, the system gains the reward $\rho(s_0)$ associated with the initial state and moves to a new state (say, $s1$), gaining also the reward $\iota(s_0, s_1)$. The cumulated reward when the system enters state $s1$ is $\rho(s_0) + \iota(s_0, s_1)$. At step 2, it gains the reward $\rho(s_1)$ associated with state $s1$, and then exits it gaining also the reward associated with the chosen transition,

[1] The adoption of an underlying Markov model implies that the modeled system meets, with some tolerable approximation, the Markov property, according to which the probability distribution of future states depend only on the current state.

and so on. In summary, the state reward is acquired if the D-MRM resides in state s_i for one time step. The reward associated with a transition $\iota(s_i, s_j)$ is gained as the process makes an instantaneous move from state s_i to state s_j. A state $s \in S$ is said to be an *absorbing state* if $P(s,s) = 1$. If a D-MRM contains at least one absorbing state, the D-MRM itself is said to be an *absorbing D-MRM.* If the absorbing states are reachable, in any number of time steps, from transient ones, it can be shown that any execution will eventually be absorbed with probability 1 (as proved for DTMCs in [29]). We assume D-MRMs to be well-formed, i.e. all states are reachable from the initial state and for all non absorbing states it is possible to reach a least one absorbing state.

Transitions can be defined through a matrix P where $P(s_i, s_j)$ represents the probability associated with the transition from state s_i to state s_j. Let us now consider two distinct states s_i and s_j. The probability of moving from s_i to s_j in 2 steps is $\sum_{s_x \in S} P(s_i, s_x) \cdot P(s_x, s_j)$. Generalizing to a k-steps path and recalling the definition of matrix product, the probability of moving from any state s_i to any other state s_j in k steps corresponds to the entry (s_i, s_j) of the matrix P^k. As a natural generalization, we can define P^0 (representing the probability of moving from a state s_i to a state s_j in 0 steps) as the identity matrix, whose elements are 1 iff $s_i = s_j$ [15,29].

Variability can be modeled quite simply in D-MRMs. We assume that variability does not affect the structure of the models, only parameters. In our case, it only affects the possible values used to label transition probabilities and rewards. This is usually expressive enough to accommodate changes in the environment that affect our system.

2.3 Probabilistic Computation Tree Logic

Formal languages to express properties of systems modeled through DTMCs have been studied in the past and several proposals are supported by model checkers to prove that a model satisfies a given property. In this paper, we focus on Probabilistic Computation Tree Logic (PCTL) [19,2], a logic that can be used to express a number of interesting reliability properties.

PCTL is defined by the following syntax:

$$\begin{aligned} \Phi &::= true \mid a \mid \Phi \wedge \Phi \mid \neg \Phi \mid \mathcal{P}_{\bowtie p}\ (\Psi) \\ \Psi &::= X\Phi \mid \Phi U^{\leq t}\Phi \end{aligned}$$

where $p \in [0,1]$, $\bowtie \in \{<, \leq, >, \geq\}$, $t \in \mathcal{N} \cup \{\infty\}$, and a represents an atomic proposition. The temporal operator X is called *Next* and U is called *Until.* Formulae generated from Φ are referred to as *state formulae* and they can be evaluated to either true or false in every single state, while formulae generated from Ψ are named *path formulae* and their truth is to be evaluated for each execution path.

The satisfaction relation for PCTL is defined for a state s as:

$$\begin{aligned}
&s \models true \\
&s \models a \quad \text{iff} \quad a \in L(s) \\
&s \models \neg\Phi \quad \text{iff} \quad s \not\models \Phi \\
&s \models \Phi_1 \wedge \Phi_2 \quad \text{iff} \quad s \models \Phi_1 \ and \ s \models \Phi_2 \\
&s \models \mathcal{P}_{\bowtie p}(\Psi) \quad \text{iff} \quad Pr(s \models \Psi) \bowtie p
\end{aligned}$$

A complete formal definition of $Pr(s \models \Psi)$ can be found in [5]; details are omitted here for simplicity. Intuitively, its value is the probability of the set of paths starting in s and satisfying Ψ.Given a path π, we denote its i-th state as $\pi[i]$; $\pi[0]$ is the initial state of the path. The satisfaction relation for a path formula with respect to a path π originating in s ($\pi[0] = s$) is defined as:

$$\begin{aligned}
&\pi \models X\Phi \quad \text{iff} \quad \pi[1] \models \Phi \\
&\pi \models \Phi_1 U^{\leq t} \Phi_2 \quad \text{iff} \quad \exists\, 0 \leq j \leq t \\
&(\pi[j] \models \Phi_2 \wedge (\forall 0 \leq k < j\, \pi[k] \models \Phi_1))
\end{aligned}$$

From the *Next* and *Until* operators it is possible to derive others. For example, the *Eventually* operator (often represented by the $\diamond^{\leq t}$ symbol) is defined as:

$$\diamond^{\leq t}\phi \ \equiv \ true\ U^{\leq t}\phi$$

It is customary to abbreviate $U^{\leq\infty}$ and $\diamond^{\leq\infty}$ as U and $\diamond$, respectively

PCTL can naturally represent reliability-related properties for a DTMC model of the application. For example, we may easily express constraints that must be satisfied concerning the probability of reaching absorbing failure or success states from a given initial state. These properties belong to the general class of *reachability properties*. Reachability properties are expressed as $\mathcal{P}_{\bowtie p}(\diamond\, \Phi)$, which expresses the fact that the probability of reaching any state satisfying Φ has to be in the interval defined by constraint $\bowtie p$. In most cases, Φ just corresponds to the atomic proposition that is true only in an absorbing state of the DTMC. In the case of a failure state, the probability bound is expressed as $\leq x$, where x represents the upper bound for the failure probability; for a success state it would be instead expressed as $\geq x$, where x is the lower bound for success. PCTL is an expressive language through which more complex properties than plain reachability may be expressed. Such properties would be typically domain-dependent, and their definition is delegated to system designers.

Recalling our example of Figure 1, we may have the following reliability requirements:

- **R1**: *"The probability of successfully handling a request is greater than* 0.999*"*
- **R2**: *"The probability for a request of being dropped by the file server of the database server because of too many concurrent connections is less than* 0.001*"*
- **R3**: *"The probability for a request of experiencing an error HTTP 503 (e.g., too many incoming requests) is less than* 0.001*"*

– **R4**: *"The probability for a request of dynamic content of experiencing a cache miss is less than* 0.25*"*
– **R5**: *"The probability for a request of static content of experiencing a cache miss is less than* 0.15*"*

These requirements can be translated into PCTL as shown in Table 1, where the notation $s = n$ refers to the identification of state n according to Fig. 1. Notice that these requirements have different sets of initial states: **R1-3** must be evaluated starting from state 0 (i.e., $S_0 = \{0\}$) while **R4-5** must be evaluated starting respectively from state 1 and 3.

Table 1. Requirements translation in PCTL

Req.	PCTL
R1	$P_{\geq 0.999}(\mathit{true}\ U\ s = 8) = P_{\geq 0.999}(\diamond\ s = 8)$
R2	$P_{\leq 0.001}(\mathit{true}\ U\ s = 9) = P_{\leq 0.001}(\diamond\ s = 9)$
R3	$P_{\leq 0.001}(X\ s = 7)$
R4	$P_{\leq 0.25}(\mathit{true}\ U\ s = 6)$ in $s = 1$
R5	$P_{\geq 0.15}(X\ s = 4)$ in $s = 3$

Given the formalisms explained so far, we can introduce in the next section the proposed approach which allow to efficiently verify non-functional requirements such as $R1 - 5$ at run-time via synthesis of symbolic expressions.

2.4 Extending PCTL with Rewards

R-PCTL is a logic language to express properties of a D-MRM. it is defined as follows [25]:

$$\begin{aligned}
\Phi &::= \mathit{true} \mid a \mid \Phi \wedge \Phi \mid \neg\, \Phi \mid \mathcal{P}_{\bowtie p}\ (\Psi) \mid \mathcal{R}_{\bowtie r}\ (\Theta) \\
\Psi &::= X\ \Phi \mid \Phi\ U\ \Phi \mid \Phi\ U^{\leq t}\ \Phi \\
\Theta &::= I^{=k} \mid C^{\leq k} \mid \ \diamond \Phi
\end{aligned}$$

Formulae Φ are named *state formulae* and can be evaluated over a boolean domain (true, false) in each state. Formulae Ψ are named *path formulae* and describe a pattern that can be matched over the set of all possible paths originating in a given state. Symbol $\bowtie$ stands for a relational operator in the set $\{\leq, <, \geq, >\}$, $p \in [0, 1]$ is a probability bound, $r \in \mathbb{R}_{\geq 0}$, and $k \in \mathbb{Z}_{\geq 0}$. $\mathit{trueU\Phi}$ can be shortened by the *eventually* operator $\diamond\Phi$, with exactly the same semantics. The expressions defined by Θ support the specification of *reward patterns*.

Let us now informally discuss the semantics of R-PCTL, first ignoring reward formulae. The intuitive meaning of the formula $\mathcal{P}_{\bowtie p}(\Psi)$ evaluated in a state s, where Ψ is a path formula, is: the probability for the set of paths originating from s and satisfying Ψ meets the bound expressed as $\bowtie p$. More precisely, the satisfaction relation for (non-reward) state formulae is defined for a state s as:

$$\begin{aligned}
&s \models true\\
&s \models a \quad \text{iff} \quad a \in L(s)\\
&s \models \neg\Phi \quad \text{iff} \quad s \not\models \Phi\\
&s \models \Phi_1 \wedge \Phi_2 \quad \text{iff} \quad s \models \Phi_1 \; and \; s \models \Phi_2\\
&s \models \mathcal{P}_{\bowtie p}(\Psi) \quad \text{iff} \quad Pr(s \models \Psi) \bowtie p)
\end{aligned}$$

A formal definition of how to compute $Pr(s \models \Psi)$ is presented in [5]. The intuition is that its value corresponds to the probability of taking a path that satisfies Ψ, among all the, possibly infinite, paths originating in s. The satisfaction relation for a path formula with respect to a path π originating in s ($\pi[0] = s$) is defined as:

$$\begin{aligned}
&\pi \models X\Phi \quad \text{iff} \quad \pi[1] \models \Phi\\
&\pi \models \Phi U \Psi \quad \text{iff} \quad \exists j \geq 0.(\pi[j] \models \Psi \wedge (\forall 0 \leq k < j.\pi[k] \models \Phi))\\
&\pi \models \Phi U^{\leq t} \Psi \quad \text{iff} \quad \exists 0 \leq j \leq t.(\pi[j] \models \Psi \wedge (\forall 0 \leq k < j.\pi[k] \models \Phi))
\end{aligned}$$

Let us now focus on the semantics of the rewards fragment of R-PCTL. We intuitively define how a state s can satisfy a formula $\mathcal{R}_{\bowtie r}$ (Θ) depending on the way the reward expression Θ is formulated.

- $\mathcal{R}_{\bowtie r}(I^{=k})$ is true in state s if the expected state reward to be gained in the state entered at step k along the paths originating in s meets the bound $\bowtie r$.
- $\mathcal{R}_{\bowtie r}(C^{\leq k})$ is true in state s if, from state s, the expected reward *cumulated* after k steps meets the bound $\bowtie r$.
- $\mathcal{R}_{\bowtie r}(\diamond\Phi)$ is true in state s if, from state s, the expected reward cumulated before a state satisfying Φ is reached meets the bound $\bowtie r$.

The third construct can be used, for example, to state the global costs of the running systems, that is, until the execution reaches a *completion* state, usually modeled by an absorbing state because of its definitive nature.

A formal semantics for the reward fragment of R-PCTL can be found in [25]. Intuitively, the expected reward $\mathcal{R}(\Theta)$ for all possible paths exiting a given state s and satisfying the pattern Θ can be computed as the sum of the rewards for each path of those paths, weighted by the probability for that path to be taken. Even in case the set of paths originating from s is infinite, the resulting infinite series can be proved to converge [5]. Notice that the probability for a path to be taken is the joint probability of all its transitions to fire, which can be computed as the product of the probabilities associated with the transitions thanks to the Markov assumption[29]. We now need to define how the expected value X for the reward can be computed for a given path $\omega = s_0 s_1 s_2 \ldots$ of the D-MRM and for a given pattern:

$$X_{I=k}(\omega) = \rho(s_k) \tag{2}$$

$$X_{C\leq k}(\omega) = \begin{cases} 0 & \text{if } k = 0 \\ \sum_{i=0}^{k-1} \rho(s_i) + \iota(s_i, s_{i+1}) & \text{otherwise} \end{cases} \tag{3}$$

$$X_{F\Phi}(\omega) = \begin{cases} 0 & \text{if } s_0 \models \Phi \\ \infty & \text{if } \forall i \;\, s_i \not\models \Phi \\ \sum_{i=0}^{min\{j|s_j \models \Phi\}-1} & \\ \quad \rho(s_i) + \iota(s_i, s_{i+1}) & \text{otherwise} \end{cases} \tag{4}$$

In Section 3 we will show how the value of X_{Θ} can be computed with algebraic techniques taking into account the presence of both numeric values and variable parameters in the D-MRM model.

Exploiting rewards we are able to express more complex requirements which may consider for example costs or latencies. Let us recall our example of Figure 1 and let us imagine to deploy the system as follows. Let us imagine to have a separate machine for each server. In particular let us imagine the scenario in which we deploy the database and the file server on a Cloud infrastructure in which bandwidth and space are billed (e.g., Amazon Simple Storage Service[2]). In this setting we are now able to interpret the couple of numbers associated to each state in Figure 1. The first number indicates the average latency, in seconds, needed to process the request including the network latency. The second number indicates the average cost for each request for being processed by the state. For example the database server state has an average cost for each request equal to 0.07$. Given these details we may express requirements such as:

– **R6**: *"The average cost for the system is less than* 0.03 $ *for each request"*
– **R7**: *"The average response time for a given request is less than* 0.022*s"*

These requirements can be translated into R-PCTL as shown in Table 2.

Table 2. Requirements translation in R-PCTL

Req.	R-PCTL
R6	$R_{\leq 0.03}(true\, U\, 7 \leq s \leq 9)$
R7	$R_{\leq 0.022}(true\, U\, 7 \leq s \leq 9)$

3 Probabilistic Verification at Runtime: Existing Approaches

Standard verification techniques for PCTL properties over DTMCs are not suitable, in general, for runtime analysis because of the intrinsic time constraints

[2] `http://aws.amazon.com/s3/`

required by solvers. Some approaches have brought state-of-the-art probabilistic model-checkers at runtime [7], providing a suitable infrastructure for many applications. Nonetheless these approaches are not general enough for at least two reasons.

Notice that, the complexity of verification can be too high in case of large systems to make the analysis meet its time constraints [9,22]. Second, analysis procedures may be unsuitable for low power devices where the large number of operations required for mathematical iterative algorithms commonly used by model-checkers may result in excessive time and energy consumption.

Model-checking can be improved in many situations both in terms of analysis algorithms, e.g. by applying space-reduction techniques (e.g. [23,3]), and via the reuse of previous results, thus opening the way for incremental analysis [26].

Besides improving standard model-checking procedures, a different approach have recently gained relevance for runtime analysis. In its seminal work [11], Daws describes a procedure for parametric probabilistic model-checking of a subset of PCTL over DTMCs. This result trod an effective path for bringing probabilistic verification at runtime, by allowing to split the analysis process in two steps. The first consists in the parametric analysis of the model with respect to the desired property, whose result is a closed mathematical expression depending on the symbolic variables appearing in the model. This step is quite complex in terms of computational time, but it can be accomplished once for all at design-time, when time is usually not a strong constraint. At runtime all that is needed to obtain the actual analysis response is to replace the symbolic variables with the actual values provided from modeling, as soon as they are discovered. The evaluation of a mathematical expression, is in general a much simpler task than model-checking, and can be performed in a very short time even on low power devices, as we shown in [12].

The main focus of this section is on parametric probabilistic verification of PCTL properties over DTMCs. In Section 3.1 we will introduce the algorithm of Daws and the subsequent improvements and implementations. In Section 3.2 we present an alternative method for parametric analysis which overcomes the limitations of Daws' algorithm, covering the entire family of PCTL formulae with an improved performance.

3.1 State Elimination Algorithms

The first approach for parametric model-checking of DTMCs has been proposed in [11]. The main contribution of that seminal work concerned the synthesis of parametric closed formulae through a *state elimination algorithm*, analogous to the one used in automata theory to synthesize regular expressions from finite state automata [20].

More precisely, Daws' algorithm allows to compute a closed mathematical expression corresponding to the probability of reaching a set of target states in any number of steps. In terms of PCTL, this corresponds to computing $Pr(true\, U\, \phi)$, with the further constraint that ϕ can only be a boolean combination of atomic formulae, i.e. it has to be possible to identify the set of states in which ϕ holds at design time and this set is not going to change at runtime. We will refer to this family of reachability formulae as *flat*.

In the next section we will introduce Daws' algorithm for reachability analysis. Afterward, in Section 3.1, we will cover its extension for the analysis of a subset of rewards formulae.

Flat Reachability Analysis. The core idea of Daws' algorithm is to consider the probability values labeling DTMC transitions as letters of an alphabet. Under this interpretation the DTMC can be seen as a finite state automaton for which it can be synthesized a variant of the regular expressions by adapting the well known state elimination algorithm [20]. Such variants of the regular expressions are named stochastic regular expressions (SREs)[11] and can be evaluated to rational mathematical expressions. The construction of SREs corresponding to the evaluation of flat reachability formulae on a DTMC is addressed by the first part of this section.

Given a flat reachability formula *true* U ϕ and a DTMC D, it possible to identify the set of states T of D that satisfy ϕ. We will call these states *target* states.

In order to simplify the exposition, let us assume for now that all the target states are absorbing. We will later relax this assumption.

We also assume the model to be well-formed, meaning that all the states are reachable from the initial state s_0. We can also prune out all the states (and the corresponding transitions) from which it is not possible to reach any of the target states. The model we obtain may no longer be a DTMC, since the elimination of a subset of the transitions may lead to sub-stochastic states, i.e. the sum of the outgoing probabilities is lower than 1, nonetheless the reduced model preserves all the information needed for the computation of the reachability formulae (a proof of correctness can be found in [18]).

Daws' algorithm consist in eliminating all the states of the reduced model but the targets and the initial state. A state elimination step is described in Figure 2. When eliminating state s, the algorithm considers all the pairs (s_i, s_j) where s_i is a direct predecessor of s and s_j is a direct successor of s. When eliminating s, the transition probability from s_i to s_j is increased by a term representing the probability of reaching s_j from s_i through s. Such a term is, roughly, the sum of the probabilities of all the possible paths, that can be computed by iterating on the length k of a path:

$$\sum_{k=0}^{\infty} p_a p_c^k p_b = \frac{p_a p_b}{1 - p_c} \tag{5}$$

The state elimination terminates when the model is composed of the initial state s_0 directly connected to each of the target states. Each of these transitions will be labeled by an SRE representing the probability of reaching the specific target state. The value of $Pr(\textit{true}\ U\ \phi)$ is just the sum of all those SRE. As it can be guessed from Figure 2, an SRE is essentially a rational expression, whose

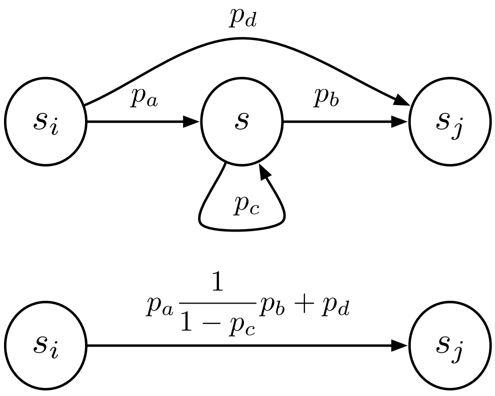


Fig. 2. SRE synthesis algorithm

numerator and denominator are polynomials having as variable the labels of DTMC transitions.

In order to generalize the approach to deal with transient target states too, it suffices to pre-process the DTMC by making all the target states absorbing. Indeed, a formula $(true\ U\ \phi)$ is satisfied by a path as soon as it firstly reaches any of the states in which ϕ holds, hence its satisfiability is not affected if the states in which ϕ holds are made absorbing. Turning a transient state into absorbing could make other states unreachable from s_0; such unreachable states have to be pruned out in order to regain a well-formed model.

In [18], Daws' algorithm has been implemented in the tool PARAM. An effective improvement provided by PARAM to the original algorithm of [11] consists in replacing the transition labels corresponding to numeric transitions by their actual value after each state elimination. This allows to exploit arithmetic simplification of the intermediate SREs that can significantly speed-up both memory consumption and subsequent mathematical operations due to state eliminations, as shown in [18].

The result of executing PARAM is a closed rational expression having as variable only the symbolic parameters of the model, since numeric ones have been already evaluated by the tool. Such an expression is then evaluated at runtime.

In our example, requirement **R1** is formalized through a flat reachability property. Its parametric verification a design-time produces the following expression:

$$\begin{aligned} Pr(true\ U\ s=8) = {} & 1 - y - 0.7 \cdot w + 0.7 \cdot x \cdot w + 0.144375 \cdot z \cdot k + 0.7 \cdot y \cdot w \\ & + -0.7 \cdot y \cdot x \cdot w - 0.144375 \cdot k + 0.144375 \cdot y \cdot k \\ & -0.144375 \cdot y \cdot z \cdot k \end{aligned}$$

When the actual values of parameters x, y, w, z, and k become available at runtime, it would suffice to substitute them in the previous expression to obtain the probability of reaching state 8. If the obtained result is ≥ 0.999, then **R1** is satisfied. Otherwise it is not.

Cumulative Rewards Analysis. A second major contribution of PARAM with respect to Daws' algorithm is its extension to deal with D-MRMs. Given as input a D-MRM D and a set of target states T, PARAM is able to compute the expected cumulative reward until a state in T is reached. The precise semantic of this measure has been provided in Equation (4).

The algorithm is again based on the state elimination procedure. Considering the pairs (s_i, s_j) of direct predecessors and direct successors of a state s, respectively, the goal is to obtain the transition reward $\iota(s_i, s_j)$ for the new transition from s_i to s_j after eliminating s. A step of this state elimination procedure is described in Figure 3, where a label p/r represents either the transition probability and the transition reward ι or the state name and its state reward ρ, respectively.

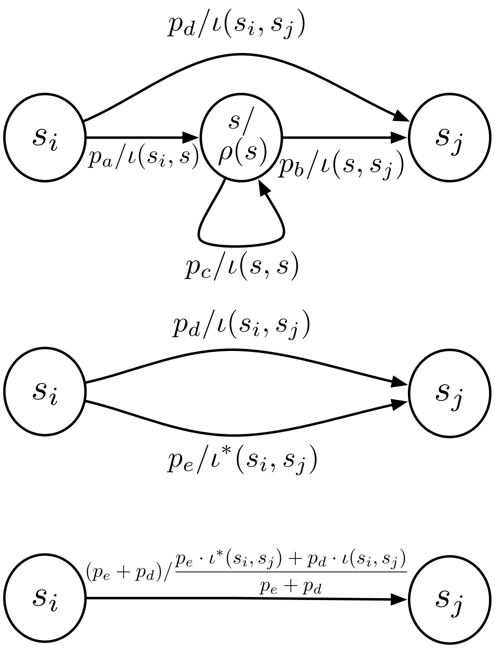


Fig. 3. State elimination for D-MRM

The value of p_e is computed as for reachability analysis as $\frac{p_a p_b}{1-p_c}$, while the value of r_e can be computed as the sum of the reward accumulated over all the possible paths from s_i to s_j through s as (with respect to the path length k):

$$\begin{aligned} r_e &= \sum_{k=0}^{\infty}(p_a p_c^k p_b) \cdot (\iota(s_i, s) + \rho(s) + (\rho(s) + \iota(s,s)) \cdot k + \iota(s, s_j)) \\ &= \iota(s_i, s) + \rho(s) + \iota(s, s_j) + \frac{p_c}{1-p_c}(\rho(s) + \iota(s,s)) \end{aligned} \tag{6}$$

The proof of correctness of the algorithms in this section can be found in [18].

As for reachability analysis, the resulting expected accumulated reward is again a rational expression with numerator and denominator being polynomials having as variables the symbolic parameters of the D-MRM, whether transition probabilities or rewards. The evaluation of such an expression at runtime requires just to replace the parameters with the numeric values coming from monitors, providing a far more efficient verification than model-checking.

In our example, **R6** requires the computation of an expected cumulated cost of a transaction. Its parametric analysis at design time produces the following expression:

$$\begin{aligned} X_{F(7 \le s \le 9)} = {} & 0.03810625 + 0.028 \cdot y \cdot x - 0.028 \cdot x - 0.03810625 \cdot y \\ & -0.01010625 \cdot z + 0.01010625 \cdot y \cdot z \end{aligned}$$

Design-Time Complexity. SRE can easily become very long and costly to manipulate. Indeed, analogously to regular expressions on finite state automata, the size of a SRE can grow as $n^{\Theta(log n)}$, where n is the number of states of the DTMC [16]. Such long expressions may take time to be manipulated at each state elimination step and may require a high memory consumption when the size of the model growths. The number of state elimination steps are in the order of $\Theta(n^3)$, as it can be easily proved [18], but the actual time each of them takes heavily depends on the complexity of the mathematical operation to be perfomed to combine SREs.

Though in the worst case $n^{\Theta(log n)}$ constitutes a complexity lower bound for computing SREs, in realistic software models most of the transitions can be assumed to be labeled by numeric values. Hence, by exploiting this assumption, instead of computing the full SRE taking transition labels as literals, PARAM intertwines the state elimination algorithm and the partial evaluation of numerical terms appearing in SREs. In other terms, at each step of the state elimination algorithm, the numeric labels are treated as numbers, thus allowing for the arithmetic simplification of intermediate results.

This induces a significant saving in the size of intermediate rational function representations, and hence an improvement in the actual computation time, as empirically shown in [18].

As a final remark, notice that the synthesis of the final rational expression may go through a large number of intermediate steps. In order to avoid any loss of accuracy, PARAM uses infinite precision rational numbers instead of double precision.

3.2 Linear Algebra Approaches

As shown in the previous section, PARAM is able to deal with reachability formulae and expected accumulated rewards, which are two subset of the properties that can be expressed in (R-)PCTL, though the most commonly used.

In this section we illustrate an approach, named WorkingMom (WM), able to deal with the entire (R-)PCTL to obtain a set of parametric closed formula through the use of linear algebraic algorithms. We will firstly show how to compute flat reachability formulae and then generalize the approach to cover the entire PCTL. Afterwards, we will present the algorithms to deal with the reward fragment of R-PCTL.

Flat Reachability Analysis. We start by focusing on flat reachability formulae for absorbing states. Recalling the structure of the transition matrix for an absorbing DTMC given in Equation (1), the matrix $I-Q$ (where I is the identity matrix of the same size of Q) has an inverse N and $N = I + Q + Q^2 + Q^3 + \cdots = \sum_{i=0}^{\infty} Q^i$ [29]. Recall from Section 2.1 that an entry q_{ij} of the matrix Q represents the probability of moving from the transient state s_i to the transient state s_j in exactly one time step. The entry n_{ij} of N represents the number of times the Markov process is expected to visit the transient state s_j before being absorbed, given that it started from state s_i. Notice that a Markov process is considered absorbed when it reaches any of the absorbing states. Notice that $Q^n \to 0$ when $n \to \infty$ (as discussed in Section 2.1), thus after enough time the process will always eventually be absorbed, no matter which state it started in.

Every time the process accesses a transient state s_i, it has a probability of being absorbed in the next time step in the absorbing state s_j given by the entry r_{ij} of the matrix R. Generalizing to all the pairs (s_i, s_j) where s_i is transient and s_j is absorbing, we can get the absorbing distribution B of the DTMC as:

$$B = N \times R$$

An entry b_{ij} of the matrix B represents the probability for the process of being eventually absorbed in s_j (in any number of states), given that it started from s_i. B is by construction a $t \times r$ matrix, where t is the number of transient states and r the number of absorbing ones.

Given a DTMC D and a set T of target absorbing states, the probability of reaching T from the initial state s_0 can be computed as:

$$Pr(\mathit{true}\ U\ T) = \sum_{s_j \in T} b_{0j} \tag{7}$$

The goal of design-time pre-computation is to compute the value of Equation (7).

Depending on the size of the system and the availability of a parallel execution environment the computation of the matrix B can be performed in different ways. In [12] we introduced a computational approach entirely based on matrix operations that can be straightforwardly suitable for parallelization. In this paper we will instead focus on how to efficiently compute matrix B in a sequential environment.

An entry b_{ij} can be computed, by definition of matrix product, as:

$$b_{ij} = \sum_{k=0..t-1} n_{ik} \cdot r_{kj} \tag{8}$$

Entries r_{ij} are readily available from matrix R. Entries n_{ik} belongs instead to the i-th row of matrix N, that is the inverse of $I - Q$.

By recalling the definition of inverse of a generic square matrix A, we know that $A \cdot A^{-1} = I$. Thus, if we are interested in computing the i-th column of the matrix A^{-1} we can simply solve the following linear system of equations:

$$A \cdot v = e_i \tag{9}$$

where e_i is the i-th column of the identity matrix, i.e. a column vector having all zero elements but for the i-th that is 1, and v is the unknown vector corresponding to the i-th column of A^{-1}. Since in Equation (8) we are required to know the entries of the i-th row of the matrix $N = (I - Q)^{-1}$, we can exploit a property of the transpose of invertible matrices, namely $(A^{-1})^T = (A^T)^{-1}$), to compute those entries.

Indeed, we are interested to the i-th row of $(I - Q)^{-1}$, which is equal to the i-th column of $((I - Q)^{-1})^T)$, which is in turn equal to the i-th column of $((I - Q)^T)^{-1}$, by the just mentioned property.

The problem of calculating the a row of the matrix N and, through (8), of B can be reduced to the solution of a linear system of equations. This solution may take a long time to be performed by using out of the shelf algorithms. Though the peculiarities of many DTMC classes, such as the models derived from software artifacts, can be effectively exploited to improve the design time efficiency, as it will be later discussed in Section 3.2.

The solution of (8) leads again to the generation of a closed rational expression, equivalent to the one computed by means of PARAM. This expression can then be brought at runtime for efficient evaluation as soon as monitors provide the actual values for symbolic parameters.

Analogously to Section 3.1, in order to generalize the procedure to the reachability of transient states it is sufficient to pre-process the model by making the target transient states absorbing. As already said, this operation may make some of the states of the DTMC unreachable from s_0. The unreachable states have to be pruned to obtain a well-formed model.

Extending to Entire PCTL. Flat reachability is the most widely used type of PCTL properties [17]. Nonetheless there are relevant requirements that cannot be easily expressed in terms of flat reachability formulae.

In this section we will incrementally show how to handle the entire PCTL by means of the WM approach. We will start by extending the reachability approach to the case of generic flat until formulae $\mathcal{P}_{\bowtie p}\ (\phi_1\ U\ \phi_2)$, where ϕ_1 is no longer constrained to be equal to *true*. Afterward we will present algorithms to verify the bounded operators X and $U^{\leq t}$. Finally we will relax the constraint for the inner state formulae to be flat and allow the nesting of temporal operators, thus covering the entire PCTL.

Flat Until Formulae. The core idea for analyzing generic flat until formulae is to reduce the problem to the analysis of equivalent reachability formulae, and then apply the solution procedures already seen.

Starting from a DTMC D and a flat until formula $\mathcal{P}_{\bowtie p}\ (\phi_1\ U\ \phi_2)$, we will construct a new DTMC $\bar{D}$ and a flat reachability formula upon $\bar{D}$ equivalent to the desired property of D. In order to construct $\bar{D}$ the following procedure has to be applied on D:

1. Add two absorbing states s_{goal} and s_{stop}
2. For all the states where ϕ_2 holds, remove all the outgoing transitions and put a single one (with probability 1) toward s_{goal}
3. For all the states where $\neg(\phi_1 \vee \phi_2)$ holds, remove all the outgoing transitions and put a single one toward s_{stop}.

Computing on $\bar{D}$ the flat reachability property $\mathcal{P}_{\bowtie p}$ (true $U\ s_{\text{goal}}$) provides the same result as computing the flat until probability of $\mathcal{P}_{\bowtie p}\ (\phi_1\ U\ \phi_2)$.

The rationale behind the previous procedure is that a path satisfying $(\phi_1\ U\ \phi_2)$ cannot pass from any state where neither ϕ_1 nor ϕ_2 hold (point 2) and has to eventually reach a state where ϕ_2 holds (point 1). At this point it is possible to apply the same mathematical machinery previously introduced for flat reachability of absorbing states, namely the solution of Equation (8) for the entry $b_{0\ s_{\text{goal}}}$.

As a final remark, notice that flat reachability formulae are special cases of flat until ones. They have been presented separately for the sake of simplifying the exposition.

Flat Next and Bounded Until. Let us focus now on the parametric analysis of Next and Bounded Until flat formulae.

The set of paths to be considered in order to estimate the probability of a path formula $X\ \phi$ in a state s_i is composed by all the 1-step long paths originating in s_i. Under the hypothesis of flat formulae, the truth of ϕ can be computed once for all at design time. As we stated in Section 2.1, the transition matrix P contains the probability of moving from a state to another in a single step. Hence, to compute the probability of reaching, from a state s_i, a state where ϕ holds in 1 step, the following procedure is in place:

$$Pr(X\ \phi_1) = \sum_{s_j \models \phi_1} p_{ij} \tag{10}$$

For example, applying (10) in state s_3 to verify the requirement **R5** of our example leads, as it should be easy to guess, to $1 - x$.

A similar procedure applies to the case bounded flat until. Indeed, each path originating in s_i and satisfying $\phi_1 U^{\leq t} \phi_2$, at a certain step $k \leq t$ will reach a state s_j where ϕ_2 holds, and for all the previous steps ϕ_1 has to hold. If we exploit the same construction used in the case of flat until formulae to construct the modified DTMC $\bar{D}$, each of these paths corresponds to a path in $\bar{D}$ that exactly at time step $k+1$ reaches the state s_{goal}, by construction. Being s_{goal} an absorbing state it is non going to be left by the path. Hence, we can conclude that any path of D satisfying $\phi_1 U^{\leq t} \phi_2$ corresponds to a path in $\bar{D}$ being at time $t+1$ in state s_{goal}.

The probability distribution of the states reached by a DTMC after exactly $(t+1)$ time steps can be computed by elevating the transition matrix P to the $t+1$-th power:

$$Pr(\phi_1 U^{\leq t} \phi_2) = (P^{t+1})_{s_0 s_{\text{goal}}} \tag{11}$$

Nested Formulae. We have so far restricted the analysis of PCTL formulae to what we called the flat fragment, that is, the set of formulae where the arguments of a path operator are boolean combinations of atomic propositions only. The peculiarity of flat formulae is that it is always possible at design time to identify the states where a state formula ϕ holds, and thus generate a parametric expression by means of the procedures previously exposed.

In the case of *nested* formulae, that is formulae $\mathcal{P}_{\bowtie p}$ (Ψ) where at least one sub-formula of ψ is again a path formula, some information needed to compute the desired parametric expression may only become available at runtime. For example, the set of states satisfying **R1** will be known only at runtime, because it depends on the actual values assigned to the model parameters. If for example such a state formula would appear as the right-hand operand of an until operator, it would not be possible to apply at design time the procedures exposed so far, since it would not be possible to identify the target states. Indeed, to evaluate a formula with nested $\mathcal{P}_{\bowtie p}$ $(\cdot)$ operators, so far we needed to know in which states its sub-formulae are satisfied, and this, in general, depends on the value of the model parameters. The same consideration can be applied recursively to sub-formulae of a sub-formula, until we reach a flat one that can be directly analyzed.

To deal with this issue we want to delay at runtime the evaluation of a nested formula, when all the knowledge concerning its sub-formulae has been gathered, without loosing the benefits of parametric verification.

Let us focus on until formulae. The solution previously provided is based on the construction of the modified DTMC $\bar{D}$. Such a construction requires to disconnect certain states from their successors and to connect them to either s_{goal} or s_{stop}. Then, for what has been previously explained, the resulting parametric expression would be the entry $b_{s_0 s_{\text{goal}}}$ of the matrix B computed as in (8) on $\bar{D}$. In order to delay at runtime the decision about the connection of a state to s_{goal} or to s_{stop}, all is needed is the addition of three more parameters per state. The first will be a coefficient m_i that multiplies all the elements p_{ij} of D. The second and the third are, respectively, two terms $a_{i\text{goal}}$ and $a_{i\text{stop}}$ to be put in correspondence of the entries $p_{s_i s_{\text{goal}}}$ and $p_{s_i s_{\text{stop}}}$ of the matrix P of $\bar{D}$. The three additional parameters can assume values 0 or 1, and their intuitive purpose is the following: assigning 0 to a parameter m_i disconnects state s_i from all its successors; assigning 1 to either $a_{i\text{goal}}$ or $a_{i\text{stop}}$ connects state s_i to state s_{goal} or s_{stop}, respectively.

Computing $b_{s_0 s_{\text{goal}}}$ at design time will lead to a parametric expression having as variables both the model parameters and the additional parameters m_i, $a_{i\text{goal}}$, and $a_{i\text{stop}}$ for each state s_i. At runtime, when information about the sub-formulae of a nested formula becomes available, the value of the additional parameters can be set in order to adapt the expression to reflect the convenient transformation of $\bar{D}$. Applying this procedure recursively on nested formulae allows to keep the benefits of parametric analysis, though it would require at most as many evaluations as the nesting depth of the formulae. Assuming most of the nested formulae to have just a few nesting levels, the impact on runtime complexity would still be limited. Another drawback in parametric analysis of nested formulae is that the resulting mathematical expressions could be longer than in the case of flat formulae due to the presence of more parameters, but the evaluation time is still not comparable with the execution of a model-checking routine for a system large enough.

Finally, the computation of next and bounded until nested formulae follows the same principle described for until ones, and they have to be computed on the model instrumented with the additional parameters m_i, $a_{i\text{goal}}$, and $a_{i\text{stop}}$. The adaptation of the mathematical procedure for the Next operator is a trivial exercise.

Reward Analysis. Equations (2), (3), and (4) of Section 2.4 formalize the semantics of the three specification patterns for reward formulae defined for R-PCTL. In this section we will provide mathematical algorithms for the analysis of each of them.

The following mathematical procedures are based on the notion of expected reward along a set of paths originating from a state s_i. In Section 2.4 this value has been intuitively defined as the sum of the rewards cumulated along each of those paths, weighted by the probability for that path to be taken. Since such a sum may contain infinite terms and could be unfeasible to compute directly, we need a different procedure more suitable for an efficient mathematical solution. Exploiting the Markov property and the linearity of the expected value [29], the computation of the expected reward for a (non empty) path originating in s_i can be computed as the sum of the state reward $\rho(s_i)$ gained in state s_i and the expected reward to be gained in each of the possible successors of s_i, weighted by the probability of moving toward it. Applying this observation to all the states S of a D-MRM leads to the following linear system of equations:

$$r_i = \rho(s_i) + \sum_{s_j \in S} p_{ij} \cdot (\iota(s_i, s_j) + r_j) \tag{12}$$

where r_i is the expected reward over all the paths originating in s_i.

In order to simplify the exposition, we will refer in this section only to flat R-PCTL formulae, meaning that in path formulae $\diamond\phi$, ϕ may not contain any of the occurrence of the modal operators $\mathcal{P}_{\bowtie p}(\cdot)$ and $\mathcal{R}_{\bowtie r}(\cdot)$. The extension to the nested fragment of R-PCTL can be achieved by instrumenting the D-MRM with additional parameters as it has been done previously for nested PCTL formulae. As further simplifying assumption, in this Section we will focus on state rewards only. Transition rewards can always be mapped into state rewards of a modified D-MRM automatically.

Let us start with the parametric analysis of formulae $\mathcal{R}_{\bowtie r}$ $(\diamond\phi)$. Recalling (4) and (12), we can define the computation of the expected cumulated rewards over all the paths satisfying $\diamond\phi$ and originating in a state s_i as the solution of the following linear system of equations:

$$r_i = \begin{cases} 0 \text{ if } s_i \models \Phi \\ \infty \text{ if } s_i \text{ is absorbing and } s_i \not\models \Phi \\ \rho(s_i) + \sum_{s_j \in S} p_{ij} \cdot r_j \;\; \text{otherwise} \end{cases} \tag{13}$$

The rational behind (13) is intuitive: a state s_i satisfying ϕ marks the satisfaction of the path formula $\diamond\phi$ and thus the end of the reward accumulation, on the other hand, an absorbing state that does not satisfy ϕ marks a path that will never satisfy $\diamond\phi$ and thus contribute to the accumulation of rewards as an infinite cost, as from the definition in (4).

Notice that the solution of (13) leads to a polynomial expression having as variables the model parameters, whether they label transition probabilities or state rewards. For example, the parametric verification of requirements **R7** leads to the following expression (notice that in this case we are considering as state reward the average execution time):

$$\begin{aligned} X_{F(7 \leq s \leq 9)} = {} & 0.21734375 + 0.084 \cdot y \cdot x - 0.084 \cdot x - 0.21734375 \cdot y \\ & -0.02165625 \cdot z + 0.02165625 \cdot y \cdot z \end{aligned}$$

Concerning formulae $\mathcal{R}_{\bowtie r}$ $(I^{=k})$, from (2) it can be computed as the sum of the rewards of every state reached in exactly k time steps, weighted by the probability of reaching it. Recall that the probability of reaching a state s_j from a state s_i in exactly k time steps is the entry $(p^k)_{ij}$ of the matrix P^k. Let us define the column vector $\bar{\rho} = [\rho(s_0), \rho(s_1), \rho(s_2), \ldots]$. The expected reward $X^{=k}$ can be computed for all the paths originating from a state s_i by the following equation:

$$X_{I^{=k}} = P^k \cdot \bar{\rho}\Big|_i \tag{14}$$

where $|_i$ indicates the i-th element of the resulting vector.

Finally, formulae $\mathcal{R}_{\bowtie r}$ $(C^{\leq k})$ require to compute the cumulated reward along all possible paths up to the k-th step. For the previous consideration, the expected reward gained at the j-th step is exactly $P^k \cdot \bar{\rho}$. Thus, to compute the cumulated reward up to the k-th step with $k \geq 1$ it is possible to apply the following equation:

$$X_{C^{\leq k}} = \sum_{i=0}^{k-1} P^i \cdot \bar{\rho} \Bigg\|_0 \tag{15}$$

When $k = 0$, $X_{C^{\leq k}} = 0$ by definition (3).

This section concludes the exposition of the mathematical machinery used within the WM approach. Most of the mathematical procedures exposed so far relies on the ability to efficiently and accurately solve a linear system of equations. In the next section we will briefly sketch the basics of the solution strategy currently used in the WM approach.

Design-Time Complexity. Solving linear systems of equations is a well studied mathematical problem, even though most of the computational approaches concern numerical solution and cannot deal with symbolic parameters [28]. The most popular algorithms to solve linear equation systems embedded in probabilistic model-checkers are iterative ones [30,28], which can efficiently solve even large systems with the desired precision in the final result and without requiring a large amount of memory.

In the WM approach it is no possible to adopt the same strategy because iterative methods do not deal conveniently with symbolic parameters. Indeed, the presence of unknown parameters makes hard to assess the convergence of the iterative algorithm. For this reason we adopted a *direct* method to solve the system. Direct methods have the additional benefit of not loosing precision in the results, and both parallel and sequential algorithms have been provided. More specifically we are interested in direct methods for the solution of sparse linear systems [10] because a Markov model for a software system is likely to have only a few non-zero entries for each row of the matrix Q, since a component or a task are usually designed to directly interact with only a few counterparts.

Sparsity of the linear system can be exploited to obtain a faster computation. Since [13], we implemented a solver based on structured Gaussian elimination and Markowitz pivoting [10]. Structured Gaussian elimination is a variation of the widely used method to triangularize linear systems which allows to reduce the solution of a large sparse equation system to the solution of a small dense one. This collapse can significantly reduce the size of the system to be actually solved. A core element of structured Gaussian elimination is the strategy used to select the order in which elements of the original system will be eliminated. In fact, each elimination step may reduce the sparsity of the obtained system, reducing in turn the global effectiveness of the method. This problem is known as *fill-in*. In order to reduce the fill-in during the elimination steps we adopted Markovitz pivoting as a selection strategy of the next element to be eliminated. Other strategies can be more suitable for specific cases but their discussion as well as mathematical details concerning structured Gaussian elimination and Markovitz pivoting are beyond the scope of this paper. The interested reader may refer for example to [10].

Finally, in order to avoid any loss of accuracy during intermediate computation steps, the WM solver uses infinite precision rational numbers for all the numeric values appearing in the models. All the mathematical procedures for the WM approach have been implemented in Maple 15[3].

4 Empirical Evaluation

In this section we provide an empirical evaluation of the tools presented in Section 3. For the verification we focus on the property $\mathcal{R}_{\bowtie r}\ (F\phi)$, where ϕ identifies the unique absorbing target state defined in all the test cases. We chose this property since it embeds a reachability formula that is both the most widely used in practical verification and the most complex to compute for the two tools. We are interested in evaluating the execution time of just the design time phase. The runtime verification, being just the evaluation of polynomial forms, takes a very short time even for very large parametric formulae, as discussed in [12].

We will provide a first comparative study of the two approaches with respect to two dimensions of the problem, namely the number of states and the number of symbolic parameters. Though this cannot be considered a complete comparison of the approaches, it provides a glimpse of how the their current implementations scale with respect to the two dimensions investigated and gives to the reader an insight about the actual computation time needed for parametric analysis of D-MRM models.

Beside PARAM and WM we added a graph from a modified WM where the linear system of equation is solved by means of the built-in solver of Maple 15. This would provide an evidence of the effectiveness of the chosen solution strategy for the actual WM.

All the models used for the tests are well-formed and generated randomly. Since we are interested in comparing the efficiency of verification algorithms, we disabled the pre-processing procedure implementing state-reduction algorithms for D-MRM that are enabled by default in PARAM. The same pre-processing could be implemented also for the WM, but is out of of the scope of this paper.

The execution environment is a Dual Intel(R) Xeon(R) CPU E5530 @ 2.40 GHz with 8Gb of ram, equipped with GNU Linux Ubuntu server 11.04 64bit. All the tests considered in this section did not overrun the available memory.

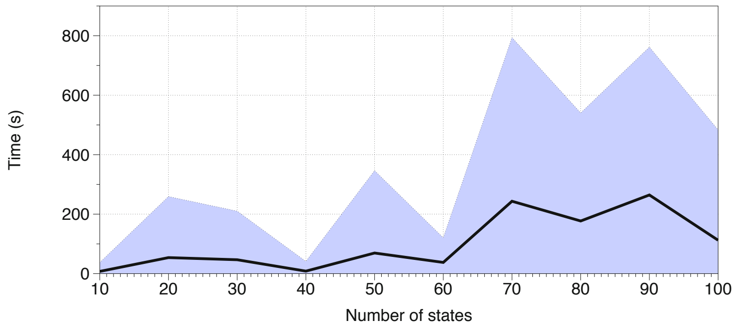
Due to the high variability in the actual execution time, we reported the average execution time with a thick black line and the maximum measured execution time in a dashed thin line.

Figure 4 reports the execution time of the two tools with respect to the number of states. All the samples have exactly 5 outgoing transitions from each transient state. There are 5 parametric transitions and 2 parametric rewards for a total of 7 symbolic parameters. The sample set is composed by 50 samples.

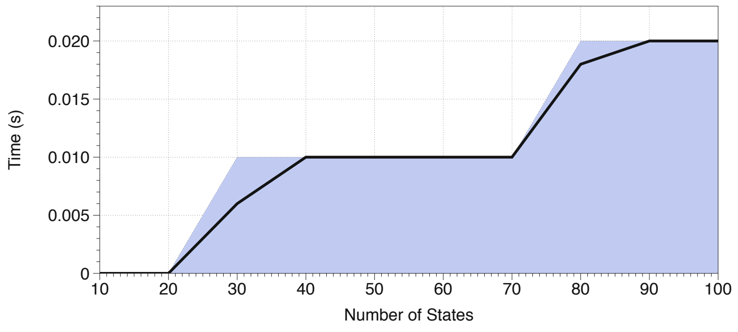
As shown in Figure 4(a), the execution time slightly grows with respect to the number of states for PARAM. Nonetheless there is a strong variability in the average execution time due to the impact on the solver of the specific topology

[3] `http://www.maplesoft.com`

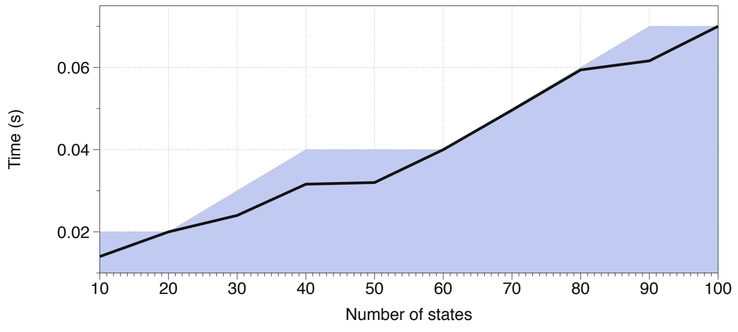
of each case. This factor will affect most of the tests proposed in this section and need further investigations to conveniently characterize input model with respect to the topology of their D-MRM. A main difference between PARAM and the approaches based on linear algebra is the order of magnitude of the



(a) PARAM

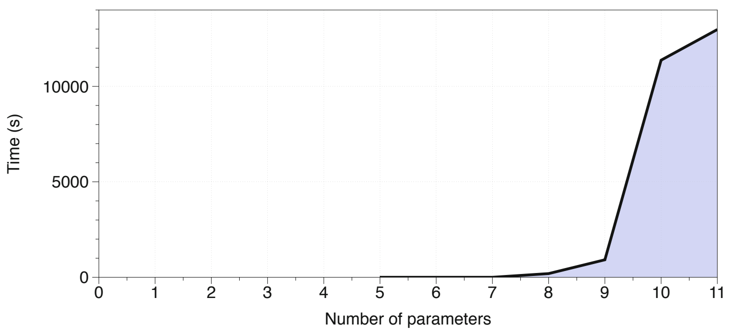


(b) Maple built-in solver

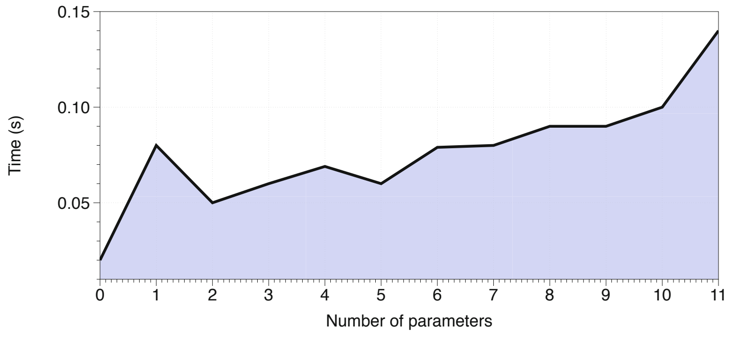


(c) WorkingMom

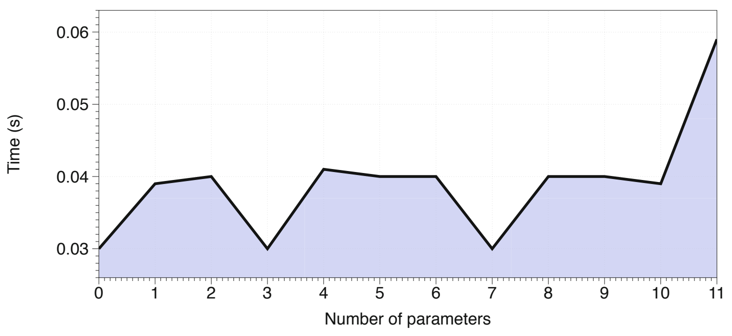
Fig. 4. Execution time vs number of states



(a) PARAM



(b) Maple built-in solver



(c) WorkingMom

Fig. 5. Execution time vs number of parameters

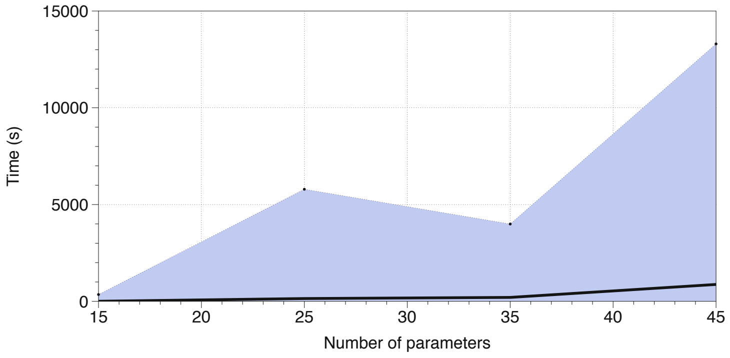
execution times, being in the first case up to 10^4 time higher. There are instead no significant differences between the Maple built-in solver and the WM.

Figure 5 shows the execution time of the three solver when the number of parameters changes. All the models have exactly 100 states, with 5 outgoing transitions per state. The parameters are distributed between transitions probability and rewards, without duplication of the same symbol.

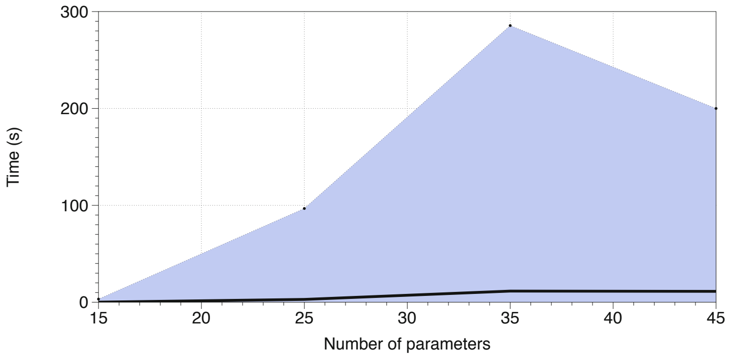
We limited the number of parameters to 10 because of the long execution time required by PARAM, as shown in Figure 5(a). In this figure, it is clear that the execution time grows sharply with the number of parameters, taking more than 2h to process models with just 10 symbolic parameters. Maple built-in solver provides a quite reasonable performance, taking no more the 0.2s in our tests, while the WM is slightly faster than this.

When the number of parameters growths up to 45, the benefits of the WM solution strategy become more visible (Fig. 6). Indeed, the built-in solver of Maple reaches a maximum execution time of about 3h, while the WM took no more than 7 minutes in the worst case. This last analysis has been performed on 200 randomly generated test cases, 50 per observation point.

Concluding, besides a global glimpse of what could be the actual execution time of state of the art tools for parametric verification of Markov models, the



(a) Maple built-in solver



(b) WorkingMom

Fig. 6. Execution time vs number of parameters

tests in this section show that the number of states and the number of parameters do not satisfactorily characterize the performance of the solver. Indeed there is a relevant distance from the maximum and the average execution time as a sign of the high variance in the measures. This suggests that specific topologies my affect, positively or negatively, the performance of the solver so far implemented, and need further investigation.

5 Conclusions and Future Work

Evolving systems require efficient verification procedure in order to timely reveal violations of their requirements. Many quantitative attribute related to the quality of service provided by the system heavily depends on environmental factors, such as the usage scenario and the interactions with external components. Those environmental factors are often out from our control and subject to unpredictable changes. A probabilistic framework could be a convenient mean to deal with this uncertainty and the availability of probabilistic model-checkers allows one to formalize and verify quantitative requirements in a fully automatic way.

Nonetheless, re-running a model-checker after any detected change may hamper the verification performance, making the system unresponsive or unable to identify the problem on time. Parametric model-checking shown to be an effective replacement for evolvable systems since it allows to partially evaluate the requirements at design time producing closed mathematical formulae quickly evaluable at runtime. The main burden of parametric model checking consists in design time computation that can be quite expensive in terms of computational time. Although the time is not supposed to be a too strict issue during design, the availability of efficient tools could speed up the entire process and provide a better interaction with the system designers.

The state of the art parametric verifiers provide reasonable performances for design-time computation, though their execution time strongly depends on the topology of each input model. This dependency has to be further investigated in order to select for each input the most efficient methodology.

Future steps in probabilistic parametric verification should focus on scalability issues that may arise in case of large models, as well as on supporting more complex models and properties, such as continuous time Markov chains that are widely used for software performance analysis. A further limitation of current tools is that they do not allow changes in the structure of the model that are not expressible as an assignment to its parameters. Overcoming this limitation could open the way to a significantly broader application in the field of self-adaptive systems.

Acknowledgments. This research has been funded by the EU, Programme IDEAS-ERC, Project 227977-SMScom and FP7-PEOPLE-2011-IEF, Project 302648-RunMore.

References

1. Andova, S., Hermanns, H., Katoen, J.P.: Discrete-time rewards model-checked. Formal Modeling and Analysis of Timed Systems, 88–104 (2004)
2. Aziz, A., Singhal, V., Balarin, F., Brayton, R., Sangiovanni-Vincentelli, A.: It usually works: The temporal logic of stochastic systems. In: Computer Aided Verification, pp. 155–165. Springer, Heidelberg (1995)
3. Baier, C., D'Argenio, P., Groesser, M.: Partial order reduction for probabilistic branching time. Electronic Notes in Theoretical Computer Science 153(2), 97–116 (2006)
4. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.-P.: Model-checking algorithms for continuous-time markov chains. IEEE Transactions on Software Engineering 29, 524–541 (2003)
5. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press (2008)
6. Baresi, L., Di Nitto, E., Ghezzi, C.: Toward open-world software: Issue and challenges. Computer 39(10), 36–43 (2006)
7. Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G.: Dynamic qos management and optimisation in service-based systems. IEEE Transactions on Software Engineering (99), 1 (2010)
8. Cheung, R.C.: A user-oriented software reliability model. IEEE Trans. Softw. Eng. 6(2), 118–125 (1980)
9. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM 42(4) (1995)
10. Davis, T.A.: Direct methods for sparse linear systems. Society for Industrial Mathematics, vol. 2 (2006)
11. Daws, C.: Symbolic and parametric model checking of discrete-time markov chains. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 280–294. Springer, Heidelberg (2005)
12. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: 33rd International Conference on Software Engineering (ICSE 2011) (accepted for publication)
13. Antonio Filieri and Carlo Ghezzi. Further steps towards efficient runtime verification: Handling probabilistic cost models. In *Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)*, Zurich, 2012.
14. Goseva-Popstojanova, K., Trivedi, K.S.: Architecture-based approach to reliability assessment of software systems. Performance Evaluation 45(2-3), 179–204 (2001)
15. Grinstead, C.M., Snell, J.L.: Introduction to Probability. Amer Mathematical Society (1997)
16. Gruber, H., Johannsen, J.: Optimal Lower Bounds on Regular Expression Size Using Communication Complexity. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 273–286. Springer, Heidelberg (2008)
17. Grunske, L.: Specification patterns for probabilistic quality properties. In: ICSE, pp. 31–40 (2008)
18. Hahn, E., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric markov models. In: Model Checking Software, pp. 88–106 (2009)
19. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal aspects of computing 6(5), 512–535 (1994)
20. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation. Addison-Wesley (2007)

21. Immonen, A., Niemelä, E.: Survey of reliability and availability prediction methods from the viewpoint of software architecture. Software and Systems Modeling 7(1), 49–65 (2008)
22. Jansen, D.N., Katoen, J.-P., Oldenkamp, M., Stoelinga, M., Zapreev, I.: How Fast and Fat Is Your Probabilistic Model Checker? An Experimental Performance Comparison. In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 69–85. Springer, Heidelberg (2008)
23. Katoen, J.-P., Kemna, T., Zapreev, I., Jansen, D.N.: Bisimulation Minimisation Mostly Speeds Up Probabilistic Model Checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 87–101. Springer, Heidelberg (2007)
24. Kramer, J., Magee, J.: Self-managed systems: An architectural challenge. In: Future of Software Engineering, FOSE 2007, pp. 259–268. IEEE (2007)
25. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic Model Checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007)
26. Kwiatkowska, M.Z., Parker, D., Qu, H.: Incremental quantitative verification for markov decision processes. In: DSN, pp. 359–370 (2011)
27. Caporuscio, M., Funaro, M., Ghezzi, C.: Architectural Issues of Adaptive Pervasive Systems. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Nagl Festschrift. LNCS, vol. 5765, pp. 492–511. Springer, Heidelberg (2010)
28. Quarteroni, A., Sacco, R., Saleri, F.: Numerical mathematics, vol. 37. Springer (2007)
29. Ross, S.M.: Stochastic Processes. Wiley, New York (1996)
30. Saad, Y.: Iterative methods for sparse linear systems. Society for Industrial Mathematics (2003)

Verification of a Self-configuration Protocol for Distributed Applications in the Cloud

Gwen Salaün[1], Xavier Etchevers[2], Noel De Palma[3], Fabienne Boyer[3], and Thierry Coupaye[2]

[1] Grenoble INP, Inria, France
`Gwen.Salaun@inria.fr`
[2] Orange Labs, France
`{xavier.etchevers,thierry.coupaye}@orange.com`
[3] UJF-Grenoble 1, Inria, France
`{Noel.Depalma,Fabienne.Boyer}@inria.fr`

Abstract. Distributed applications in the cloud are composed of a set of virtual machines running a set of interconnected software components. In this context, setting up, (re)configuring, and monitoring these applications is a real burden since a software application may depend on several remote software and virtual machine configurations. These management tasks involve many complex protocols, which fully automate these tasks while preserving application consistency. In this paper, we focus on a self-configuration protocol, which is able to configure a whole distributed application without requiring any centralized server. The high degree of parallelism involved in this protocol makes its design complicated and error-prone. In order to check that this protocol works as expected, we specify it in LOTOS NT and verify it using the CADP toolbox. The use of these formal techniques and tools helped to detect a bug in the protocol, and served as a workbench to experiment with several possible communication models.

1 Introduction

Cloud computing emerged a few years ago as a major topic in modern programming. It leverages hosting platforms based on virtualization, and promises to deliver resources and applications that are faster and cheaper with a new software licensing and billing model based on the *pay-per-use* concept. For service providers, this means the opportunity to develop, deploy and sell cloud applications worldwide without having to invest upfront in expensive IT infrastructure.

Distributed applications in the cloud are composed of a set of virtual machines (VMs) running a set of interconnected software components. However, the task of configuring distributed applications is a real burden. Indeed, each VM includes many software configuration parameters. Some of them refer to local configuration aspects (*e.g.*, pool size, authentication data) whereas others contribute to the definition of the interconnections between the remote elements (*e.g.*, IP address and port to access a server). Therefore, once it has been instantiated, each

VM has to apply a set of dynamic settings in order to properly configure the distributed application. On the whole, existing deployment solutions rarely take into account these different configuration parameters, which are mostly managed by dedicated scripts that do not work completly automatically (human intervention is needed). Moreover, these solutions are application-dependent and only work for specific distributed applications to be deployed: Google App Engine for instance only deploys Web applications whose code conforms to very specific APIs (*e.g.*, no Java threads), Microsoft Azure only supports applications based on Microsoft technologies, Salesforce only focuses on customer relationship management, etc.

In this paper, we present an abstract model for describing component-based applications and an innovative self-configuration protocol which automates the deployment of these distributed applications in the cloud. Once the VMs are instantiated, the self-configuration protocol is able to configure the whole application without requiring any centralized server and does not require a complex scripting effort. The high degree of parallelism involved in this protocol makes its design complicated and error-prone. Consequently, we decided to formally specify and verify this protocol in order to find possible bugs using state-of-the-art model checking techniques. The self-configuration protocol was specified using the specification language LOTOS NT [9] (LNT for short) and verified using CADP verification tools [15]. LNT is a simplified variant of the E-LOTOS standard [19] that combines the best features of imperative programming languages and value-passing process algebras. LNT has a user-friendly syntax, and supports the description of complex data types written using a functional specification language. Since LNT relies on classic programming paradigms, this greatly simplifies the design and analysis process, and reduces the gap between the specification and the real implementation of the system. In this work, these formal techniques and tools helped to detect a major bug in the protocol, which was corrected in the Java reference implementation. The LNT specification also served as a workbench to experiment with several possible communication models, and these experiments helped us to avoid an erroneous design.

It is worth emphasizing that the self-configuration protocol is one of the base components of a French project called OpenCloudware[1], aiming at building an open software engineering platform, for the collaborative development of distributed applications to be deployed on multiple Cloud infrastructures. OpenCloudware is a funded project that started in 2012 for three years and involves many companies and research centers in France.

The rest of this paper is organized as follows. Section 2 introduces the distributed application model and the self-configuration protocol. We present the LNT specification of the protocol in Section 3 and its verification in Section 4. After comparing our experience with related work in Section 5, we conclude this paper in Section 6.

[1] See `http://opencloudware.org` for more details.

2 Self-configuration Protocol

2.1 Application Model

The configuration of a cloud application is specified using a global model composed of a set of interconnected software components running on different VMs. A component is a runtime entity that has some configuration parameters and one or more interfaces. An interface is an access point to a component that supports a finite set of methods. Interfaces can be of two kinds: server interfaces, which correspond to access points accepting incoming method calls, and client interfaces, which correspond to access points supporting outgoing method calls. Bindings make explicit connections between components' client interfaces and server interfaces. A binding is local if the components involved in the binding are running on the same VM. A remote binding is a binding between a client interface of a local component and a server interface provided by a component located in another VM. A client interface is also characterized by a property named *contingency*, which indicates whether this interface is optional or mandatory. By extension, the contingency of a binding corresponds to the contingency of its client side. A component has also a lifecycle that represents its state (started or stopped). Finally, an application model identifies each VM belonging to the application, the set of components running on each VM, and their local/remote bindings. A simple example of application model is given in Figure 1 (left), where c stands for client and s for server.

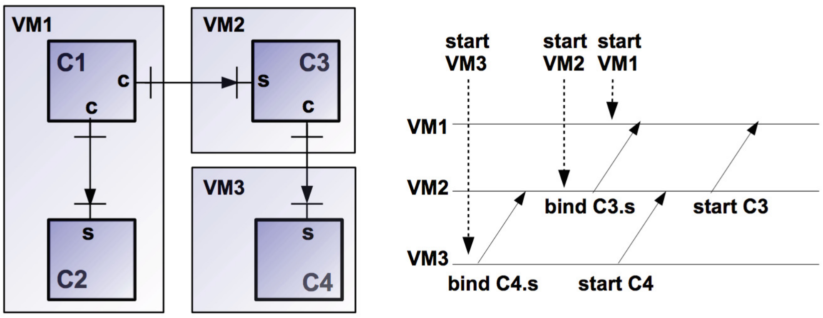


Fig. 1. Example of application configuration (left) and self-configuration protocol execution (right)

2.2 Self-configuration Principles

The configuration starts when the deployment manager instantiates all VMs. Each VM embeds a configurator which drives and encodes most of the self-configuration behaviour. A virtual machine is also equipped with two buffers (one input buffer and one output buffer) for communicating with the other VMs. All communications transit through a MOM.

Each VM embeds the application model and a configurator. It is worth observing that embedding the whole application model on each VM is not an optimal solution, particularly if the system is planned to be reconfigured. Here, we made this simplification because we focused only on deployment. An alternative solution could be to embed only on each virtual machine the information necessary for its (re)configuration, that is information about the (remote) components connected to local components.

The configurator manages the configuration of the components inside the VM, and participates in the binding configuration between components and in the application start-up. To this end, each configurator has the ability to create and configure components, send server interfaces (for binding purposes), bind component client interfaces to server ones, start components, and send messages to other VMs indicating that a local component has been started. To bind a client interface, the local configurator in charge of the component on the client side needs the corresponding server interface, that is, the required information to access to this interface (IP, port, etc.). This server interface can be local (in this case the local configurator can manage this by itself), or it can be remote (in this case the remote configurator sends the server interface to the local configurator of the corresponding remote VM).

The configurators send their server interfaces and start messages, according to the application model, through a Message Oriented Middleware [4] (MOM). MOMs implement a message buffering system that enables configurators to exchange messages in a reliable and asynchronous way. From a local point of view, each VM is equipped with two buffers, one output buffer storing messages destinated to other VMS and one input buffer storing messages coming from other VMs.

It is worth observing that, for scalability purposes, the self-configuration protocol used to configure distributed applications is *decentralized*. Once the VMs are instantiated, the self-configuration protocol is able to configure the whole application without requiring any centralized server. The self-configuration protocol is also *loosely-coupled*. Each VM starts the self-configuration protocol just after the boot sequence (instantiation of VMs by the deployment manager) without needing to know about the state of other VMs. The configuration of the distributed application will progress each time a VM belonging to the application becomes available. This avoids the need for global synchronization between VMs during the configuration protocol.

2.3 Protocol Description

The protocol execution is driven by the configurators embedded on each VM. All configurators evolve in parallel, and each of them carries out various tasks following a precise workflow that is summarized in Figure 2 where boxes identified using natural numbers (❶, ❷, etc.) correspond to specific actions (`CREATEVM`, CREATECOMPO, etc.). Diamonds stand for choices, and each choice is accompanied by a list of box identifiers that can be reached from this point.

Based on the application model, the configurator starts (❶), successively creates all the components described in the model for this VM (❷), and binds local components (❸). Note that diamonds in the workflow propose several options, because a VM may not have local bindings for instance, and in such a case, the configurator jumps to the next step. In order to set up remote bindings, both VMs need to interact by exchanging messages through the MOM (❹). For each binding associated to two components C_1 and C_2 (involved respectively in the binding between a server interface and a client interface), the configurator K_1 (responsible for C_1) sends the server interface to configurator K_2 (responsible for C_2). This server interface includes all information required by C_2 to interact with C_1, that is, when K_2 receives a message containing such an interface, it proceeds with the binding of C_2 to C_1.

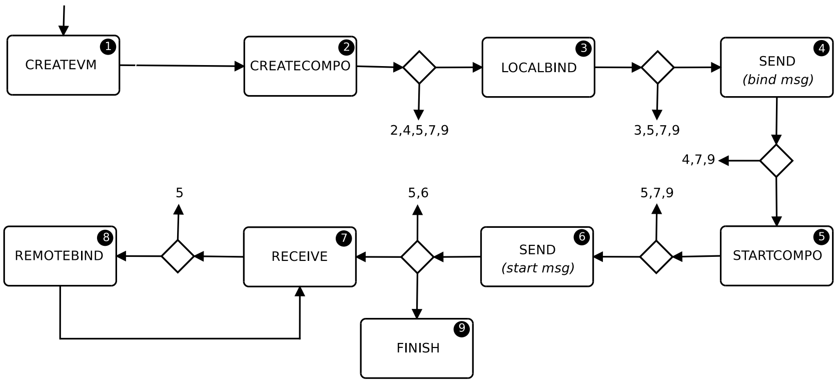


Fig. 2. Configurator workflow

Once the configurator has sent all its server interfaces, it can launch the process for starting the applicative components. The configurator first launches the local components that can be started (❺). At that moment in the protocol execution, the only components that can be started are components without mandatory client interfaces or components whose mandatory client interfaces are all connected to local components. For each component C_{server} then started, the configurator sends to every remote component connected to it through an application binding, a start message (❻) indicating to the remote component that this C_{server} component is started. When the configurator has started all the local components that can be launched, it starts reading from its input communication buffer (❼). Two kinds of message can be received: (i) upon receiving a binding request message, the configurator binds the local component to the remote one (❽), (ii) upon receiving a message indicating that a remote component has been started, the configurator keeps track of this information and goes back to ❺ in order to check whether other local components can be started (those with all mandatory client interfaces connected and corresponding server components started).

Figure 1 provides an application example (left) and the corresponding self-configuration protocol execution (right). This execution scenario shows the communications exchanged between the VM configurators to start the application. We can see that first the VM3 configurator (in charge of C4) sends a binding message with C4 server interface to VM2 configurator (in charge of C3). VM2 sends C3 server interface to VM1 configurator. Upon receptions both configurators can make these bindings effective. When VM3 starts C4, a message is sent to VM2. Upon reception, VM2 can start C3, and sends a message to VM1 indicating that C3 has been started.

2.4 Implementation

From an implementation point of view, we rely on the Open Virtualization Format [1] (OVF) in order to describe an application, that is a set of interconnected components hosted on various virtual machines. OVF is an open and extensible standard for packaging and distributing virtual appliances or software to be run in virtual machines. However, OVF is not designed for describing architectural aspects (components, interfaces, bindings). Therefore, we have proposed an extension of OVF, which aims at offering an architectural view of the distributed application which is embedded within the VMs of an OVF package. Using OVF offers higher level control abstractions, compared to specific configuration scripts existing in current industrial solutions.

Our extension of OVF enables the description of a distributed application through an XML-based description encompassing not only the notions of hardware requirements, disk images, etc., but also components, interfaces, and bindings. Adding the notion of VM to each component description (using the *virtual-node* tag) also enables the description of the distribution constraints of components within virtual machines.

The deployment engine is implemented in Java and builds upon this ADL to configure automatically an application described with this formalism. We have already used this process for deploying real applications such as Springoo, CLIF, or Tune. Springoo is a Java EE multitiered application enabling the management of markets, offers, and services in a company. CLIF [12] is a load injection framework, which provides a Java-based, open source, generic infrastructure to generate load on any kind of systems, and gather performance measurements (requests response times, computing resources usage, etc.). Tune [8] is a global autonomic management system in Java. Evaluation results show that the self-configuration protocol allows a human administrator to reduce significantly the duration for deploying a large number of applications. A more detailed description of the self-configuration protocol (description, technical details, evaluation, etc.) can be found in [13,14].

3 Specification

We specified the protocol in LNT [9], one of the input languages of the CADP verification toolbox [15]. We chose LNT as our specification language because (i) it provides expressive enough operators, in particular rich datatype descriptions, for modelling the self-configuration protocol, (ii) its user-friendly notation simplifies the specification writing, and (iii) it is equipped with state-of-the-art verification tools in order to check that the protocol respects some key-properties.

3.1 LNT in a Nutshell

LNT is a simplified variant of the E-LOTOS standard [19] that combines the best features of imperative programming languages and value-passing process algebras. LNT supports both the description of complex data types and of concurrent processes using the same user-friendly syntax. LNT formal operational semantics is defined in terms of LTSs (Labelled Transition Systems).

LNT processes are built from actions, sequential compositions (;), conditions (**if .. then .. else .. end if**), assignments (:=), looping behaviours (**loop .. end loop**), choices (**select .. [] .. end select**), and parallel compositions (**par .. || .. end par**). Communication is carried out by rendezvous on a set of synchronization actions (multiway synchronization points) with bidirectional transmission of multiple values. Synchronizations may also contain optional guards (**where**) expressing Boolean conditions on received values. Processes are parameterized by sets of actions (alphabets) and input/output data variables.

LNT specifications can be analysed using CADP, a verification toolbox that has been in continuous development since the late 80s. CADP is dedicated to the design, analysis, and verification of asynchronous systems consisting of concurrent processes interacting via message passing. The toolbox contains about 70 tools and libraries that can be used to make different analyses such as simulation, model checking, equivalence checking, compositional verification, test case generation, or performance evaluation. CADP was successfully applied to real-world and industrial case studies in many different fields such as telecommunication protocols, hardware design, embedded systems, or avionics.

In the rest of this section, we will present a few excerpts of the self-configuration protocol LNT specification.

3.2 Data Types and Functions

Data types are used to describe the distributed application model, that is, VMs, components, interfaces (client and server), bindings between components, messages, buffers, etc. We show below a few examples of data types. An application (`TApplication`) consists of a set of VMs and a set of bindings. A VM (`TVM`) consists of an identifier and a set of components. A component (`TComponent`) is characterized by an identifier, a set of client interfaces, and a set of server interfaces. A client interface (`TClient`) is a couple *(identifier, contingency)*, the contingency (`TClientType`) being either mandatory or optional.

```
type TApplication is
     tapplication (vms: TVMSet, bindings: TBindingSet)
end type
type TVMSet is set of TVM end type
type TVM is
     tvm (id: TID, cs: TComponentSet)
end type
type TComponent is
     tcompo (id: TID, cs: TClientSet, ss: TServerSet)
end type
type TClient is
     tclient (id: TID, contingency: TClientType)
end type
type TClientType is mandatory, optional end type
```

Functions apply on data expressions which describe the distributed application. These functions are necessary for three kinds of computation: (i) extracting information from the application model, (ii) describing buffers and basic operations on them, (iii) keeping track of the started components to know when another component can be started, *i.e.*, when all its mandatory client interfaces are connected to started components. Functions are also defined to check that there is no cycle of mandatory client interfaces through bindings in the application model, and that all the mandatory client interfaces are bound. Let us show, for illustration purposes, the function `add`, which adds a message `m` to a buffer `q` storing messages in a list with respect to a FIFO strategy (we add messages at the end of the buffer and read from the beginning). `TBuffer` is specified as a list of messages of type `TMessage`, equipped with classic constructors **cons** and **nil**. It is worth observing in this example that LNT uses the classic ingredients of functional programming, namely pattern matching and recursion.

```
function add (m: TMessage, q: TBuffer): TBuffer is
  case q in
  var hd: TMessage, tl: TBuffer in
       nil -> return cons(m,nil)
     | cons(hd,tl) -> return cons(hd,add(m,tl))
  end case
end function
```

3.3 Processes

They are used to specify VMs (configurator, input and output buffer), the communication layer (MOM), and the whole system consisting of VMs interacting through the MOM. Each VM consists of a configurator and two buffers, namely `bufferIn` and `bufferOut`, which store input and output messages, respectively. The configurator drives the behaviour of each VM, and encodes most of the protocol functionality. The MOM process reproduces the communication media behaviour used to make VMs interact together. The MOM is equipped with a set of FIFO buffers in order to store messages being exchanged. There is a buffer for each VM, and messages transiting by the MOM are temporarily stored in the buffer corresponding to the VM to which the message is destinated.

For illustration purposes, we present two excerpts of LNT processes. The first one is the `SELFCONFIG` process, which encodes the behaviour of the whole protocol. We give in Figure 3 an architectural view of this process with the MOM and as many instances of the configurator and buffer processes as there are VMs.

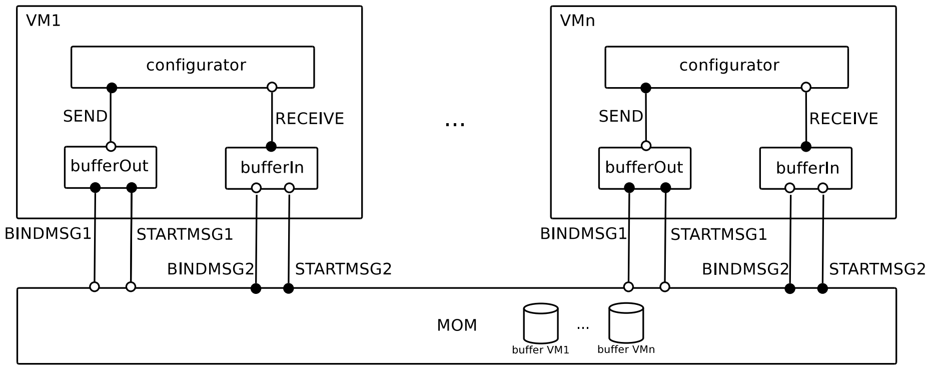


Fig. 3. Architectural view of the whole protocol

The `SELFCONFIG` process defines first the list of actions used in its behaviour (`CREATEVM`, `SEND`, etc.). Actions can be typed (with the types of their parameters), but this is optional and we use the keyword **any** in that case. This process applies on an input application defined in function `appli()`. A pair of actions (`CHECKCYCLE` and `CHECKMANDATORY`) are introduced at the beginning of the process body for verification purposes. These actions have as parameters Boolean values computed by calling functions, *e.g.*, `check_cycle_mandatory`, which indicate whether the input application respects some structural constraints, *e.g.*, absence of cycle through mandatory client interfaces.

The LNT parallel composition is expressed with the **par** construct followed by the list of actions that must synchronize (nothing for pure interleaving). The first process called in the `SELFCONFIG` process is the MOM, which is composed in parallel with the rest of the system, and synchronizes with the other processes on `BINDMSG`i and `STARTMSG`i messages (i=`1`,`2`). More precisely, the MOM has five possible behaviours: it can receive a binding (`BINDMSG1`) or a start message (`STARTMSG1`), send a binding (`BINDMSG2`) or a start message (`STARTMSG2`) if one of its buffers is not empty, or terminate (`FINISH`). Messages suffixed with `1` correspond to emissions from a VM to the MOM, and messages suffixed with `2` correspond to emissions from the MOM to a VM.

After the MOM, a piece of specification (deployment manager) is in charge of instantiating the set of VMs (`CREATEVM`). Finally, as many VMs as are present in the input application (two machines VM1 and VM2 in the specification below) are generated. Since the number of VMs depends on the application, this LNT process is generated automatically for each new application, by a Python program we wrote. Each machine consists of a configurator, which synchronizes with two local buffers (bufferIn and bufferOut) on messages SEND and RECEIVE. The two buffers as well as the MOM are initialised empty.

It is worth noting that we use two kinds of action in our specification: actions which corresponds to communications between two processes (SEND and RECEIVE for synchronizations within a VM, BINDMSG and STARTMSG for synchronizations between VMs), and actions tagging specific moments of the execution that will be useful in the next section to analyse the protocol (CHECKCYCLE, CHECKMANDATORY, CREATEVM, CREATECOMPO, LOCALBIND, REMOTEBIND, STARTCOMPO, and FINISH). For instance, termination of the protocol is made explicit by a synchronization involving all processes on FINISH. Here is an example of SELFCONFIG process (for two VMs identified by VM1 and VM2):

```
process SELFCONFIG [CREATEVM:any, SEND:any, ..] is
   var appli: TApplication in
     appli:=appli();
     CHECKCYCLE (!check_cycle_mandatory(appli));
     CHECKMANDATORY(!check_mandatory_connected(..));
     par BINDMSG1, BINDMSG2, STARTMSG1, .. in
       MOM[..](vmbuffer(VM1,nil),vmbuffer(VM2,nil))
     ||
       par CREATEVM, FINISH in
         par FINISH in (* deployment manager *)
           CREATEVM (!VM1) ; FINISH
         ||
           CREATEVM (!VM2) ; FINISH
         end par
       ||
         par FINISH in
           (* first machine, VM1 *)
           par SEND, RECEIVE, FINISH in
             configurator [..] (VM1,appli)
           ||
             par FINISH in
               bufferOut[SEND,BINDMSG1,..](nil)
             ||
               bufferIn[RECEIVE,BINDMSG2,..](VM1,nil)
             end par
           end par
         ||
           ... (* second virtual machine, VM2 *)
  end par end par end par end var
end process
```

Now we detail the bufferIn process which can synchronize with other processes (MOM and local configurator) on four actions, namely RECEIVE, BINDMSG, `STARMSG`, and `FINISH`. This process also has two data parameters corresponding to the identifier of its VM, and to the buffer storing messages. Its behaviour is a choice (**select** in LNT) among different possibilities: `bufferIn` can either (i) store messages coming from the MOM and destinated to its VM, or (ii) interact with the local configurator when the configurator decides to read from the input buffer. In the first case, two kinds of messages can be received: a request for binding (`BINDMSG`), or a message announcing that a remote component has been started (`STARTMSG`). In both cases, a message (`TMessage`) is built from the information provided as parameter to the action (*i.e.*, `csvr`, `cclt`, etc.) and stored in the local buffer (`add(..)`). In order to ensure that the input buffer receives only messages destinated to its VM, we use a LNT feature which makes messages with sent parameters synchronize only if they share common parameters. In this case, we use the VM identifier (`!vmid`, first parameter of `BINDMSG` and `STARTMSG` messages). This means that the MOM process will also use such VM identifiers as first parameter of these messages. In the second case, if the buffer is not empty, a message may be retrieved and treated by the local configurator (synchronization on `RECEIVE`).

```
process bufferIn [RECEIVE:any,BINDMSG:any,STARTMSG:any,FINISH:any]
            (vmid: TID, q: TBuffer) is
   var recip, csvr, cclt, idclt, idsvr: TID, m: TMessage in
     select
       BINDMSG (!vmid, ?csvr, ?cclt, ?idclt, ?idsvr) ;
         bufferIn [RECEIVE,BINDMSG,STARTMSG,FINISH]
           (vmid,add(bindmsg(vmid,csvr,cclt,idclt,idsvr),q))
       []
       STARTMSG (!vmid, ?csvr, ?cclt) ;
         bufferIn [RECEIVE,BINDMSG,STARTMSG,FINISH]
           (vmid,add(startmsg(vmid,csvr,cclt),q))
       []
       if not(empty(q)) then
         m:=retrieve(q); RECEIVE (!vmid, !m);
         bufferIn [RECEIVE,BINDMSG,STARTMSG,FINISH] (vmid,remove(q))
       else
         bufferIn [RECEIVE,BINDMSG,STARTMSG,FINISH] (vmid,q)
       end if
       []
       FINISH
     end select
   end var
end process
```

4 Verification

To verify the protocol, we apply the LNT specification of the protocol to a set of distributed applications to be configured. From the specification and the target application, CADP exploration tools generate an LTS describing all the possible executions of the protocol. In this LTS, transitions are labelled with the actions introduced previously, and we use these actions to check that the protocol works as expected.

4.1 Verification Tasks

We identified three facets of the protocol that must be preserved by the protocol, that are structural invariants, temporal properties, and lifecyles.

Invariants. First of all, we verify that each input application respects a few structural properties, such as *"there is no cycle in the application through mandatory client interfaces"* or *"all mandatory client interfaces are connected"*. This is checked at the beginning of the protocol using functions which extract this information from the application model given as input. These functions return Boolean values which are then passed as parameters to specific actions (`CHECKCYCLE` and `CHECKMANDATORY`). Then, we use a safety property to check that these actions do not appear in the LTS with the wrong Boolean parameter. For instance, we never want the `CHECKCYCLE` action to have a `TRUE` parameter value indicating that there is a cycle of mandatory client interfaces. This is written as follows in μ-calculus, the temporal logic used in CADP, and such properties are verified automatically using the EVALUATOR model checker [24]:

```
[ true* . "CHECKCYCLE !TRUE" ] false
```

Properties. Secondly, we use model checking techniques to verify that some key-properties are respected during the protocol execution. To do so, we formalise in μ-calculus (and check) 14 safety and liveness properties that must be preserved by the configuration protocol. Here are a few examples of these properties:

– `FINISH` is eventually reached in all paths

```
mu X . (< true > true and [ not 'FINISH' ] X)
```

– A `STARTMSG2` message cannot appear before a `STARTMSG1` message with the same parameters

```
[ true*.STARTMSG2 ?vm:String ?cx:String ?cy:String.
        true*.STARTMSG1 !vm !cx !cy ] false
```

Note that we use the latest version of EVALUATOR (4.0) which enables us to formulate properties on actions and data terms. Here for example, we relate parameters in both messages saying that the VM (`vm`) and components (`cx` and `cy`) concerned by this message must be the same.

– A component cannot be started before the components it depends on

```
[ true* . 'STARTCOMPO !.* !C1' . true* . 'STARTCOMPO !.* !C2' ] false
```

This property is automatically generated from the application model because it depends on the bindings for each component. As an example, if a component `C1` is connected through a mandatory client interface to a component `C2`, we generate the property above meaning that we will never find a sequence where `C1` is started before `C2`.

– All components are eventually started

```
( mu X . ( <true> true and [ not 'STARTCOMPO !.* !C1' ] X ) )
                              and
( mu X . ( < true > true and [ not 'STARTCOMPO !.* !C2' ] X ) )
                            and ...
```

This property is also generated because the number of components and their identifiers depend on the application model.

Lifecycles. Finally, we check that each VM behaviour isolated from the whole LTS respects the correct ordering of actions. To do so, on the one hand, we have specified an LTS corresponding to the configurator lifecycle. This LTS is obtained by flattening the workflow presented in Figure 2 and consists of 8 states and 26 transitions. On the other hand, we apply successively hiding and reduction techniques on the whole state space to keep configurator actions corresponding to a specific VM. Then, we check that the resulting LTS is included (branching preorder) into the first one (configurator lifecycle) using the Bisimulator equivalence checker [5]. For each application, we also extract the MOM behaviour and check that it is included in the LTS given in Figure 4.

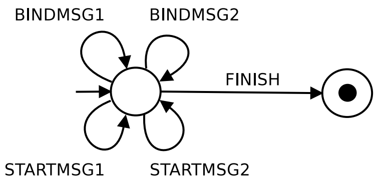


Fig. 4. LTS representing the MOM lifecycle

4.2 Experiments

They were conducted on about 150 applications, which are quite different and enabled us to check boundary cases. For instance, we used applications where components can be started in parallel (interleaving) and others where they can only be started in a very precise order. It is worth observing that, as model checking helps to fing bugs, the more examples we check, the more chances we have to find problems in the protocol.

Table 1 summarizes some of the results obtained on application examples of our dataset. Each example is characterized in terms of number of virtual machines, number of components, and number of local/remote bindings ("b." stand for bindings in the table). We give the size of the LTS generated using CADP by enumerating all the possible executions of the system, as well as the time to obtain this LTS and verify all the features presented above (checking invariants, properties, and lifecycles). The resulting LTS has been minimized using strong reduction.

Experiments have been carried out on a Xeon W3550 (3.07GHz, 12GB RAM) running Linux, and it takes about 3 days to generate and check all the examples of our database. We can see first that systems involving only a couple of virtual machines and a few remote bindings are generated and checked in reasonable time (examples 0010, 0061, and 0090 in Table 1).

Computation times and LTS sizes grow exponentially as the number of remote bindings and VMs increase. As far as remote bindings are concerned, the more bindings, the more messages exchanged among VMs. This results in large LTSs (see, *e.g.*, example 0092) which are generated quite fastly because the number of processes in parallel is reasonable (3 VMs in example 0092 for instance). In this case, verification takes time because LTSs have to be traversed exhaustively. This is also interesting to note the size and time increase when looking at examples 0086, 0087, and 0088 where by adding one remote binding, the LTS size approximately doubles as well as verification time. In contrast, we can see that the number of local bindings can be quite high without really impacting size and time verification results. Similarly, the number of components does not really affect the results (see, *e.g.*, example 0010).

If we focus now on the number of VMs, we can see that when we have systems with four VMs, LTS generation time grow exponentially (examples 0136 and 0145). This is because the number of processes evolving in parallel increase with the number of VMs and this makes the exploration step very time consuming. The resulting LTS is quite small though and its verification pretty fast.

Fortunately, our goal here was not to fight the state explosion problem, but to find possible bugs in the protocol. Most bugs do not come from the system's size, but from boundary cases where enumerative tools are very efficient by exploring all the possible execution scenarios.

4.3 Issues Identified

The specification and verification helped us to detect a major bug in the protocol and to experiment on the communication model. Firstly, there was a problem in the way local components are started during the protocol execution. After reading a message from the input buffer, the configurator must check all its local components, and start those with mandatory client interfaces bound to started components. However, one traversal of the local components is not enough. Indeed, launching a local component can make other local components startable. Consequently, starting local components must be done in successive iterations, the algorithm stops when no more components can be started. If this is not implemented as a fix point, the protocol does not ensure that all components involved in the architectue are eventually started. This bug was detected quite early during the verification process (after a few examples) thanks to one of the properties presented in Section 4.1 (all components are eventually started). It was corrected in both the specification and the Java implementation.

Secondly, there are many ways to implement the MOM. We used our specification, modifying the MOM process, to carry out experiments on how communication among VMs could be implemented (no MOM, MOM with one buffer, two

Table 1. Experimental results

	Size				LTS	Time (m:s)	
	VMs	compo.	local b.	remote b.	(states/transitions)	LTS gen.	Verification
0010	2	15	2	2	1,788/4,943	0:09	2:23
0061	2	6	3	5	5,091/18,354	0:10	1:45
0090	2	6	3	8	33,486/137,401	0:50	6:44
0092	2	6	9	10	81,822/349,319	1:20	27:20
0122	3	6	6	0	514/1,346	0:14	00:26
0038	3	5	0	4	31,334/109,315	4:01	8:15
0086	3	6	34	4	60,851/226,217	8:14	19:30
0087	3	6	34	5	153,056/645,168	14:02	49:42
0088	3	6	34	6	306,136/1,392,439	25:53	98:42
0136	4	4	0	3	3,350/11,997	84:24	1:02
0145	4	7	4	2	18,314/78,206	191:20	6:02

buffers, MOM with n buffers, $2n$ buffers, etc.). We found out that using a single buffer in the MOM is erroneous because the protocol can get momentarily stuck if a VM is not yet started, and the first message in the buffer has to be sent out to that VM. One buffer per machine is necessary to avoid these blocking issues, and this MOM structure was chosen after having carried out these experiments.

5 Related Work

The formalisms and mechanisms offered by the industrial solutions for configuring applications in the cloud are generally basic, proprietary, not exhaustive, and not extensible: they permit neither a fine-grained description of the distributed application nor the management of its deployment process. Moreover, such solutions have often important restrictions concerning:

– the programming models like Google App Engine that only deploys Web applications whose code must conform to very specific APIs (*e.g.*, no Java threads)
– the underlying technologies like Microsoft Azure that is confined to the applications based on Microsoft technologies
– the business domains they address like Salesforce that focuses on customer relationship management

A few recent projects [16,10,25] proposed languages and configuration protocols for distributed applications in the cloud. [10] adopts a model driven approach with extensions of the *Essential Meta-Object Facility* (*EMOF*) abstract syntax[2] to describe a distributed application, its requirements towards the underlying

[2] This syntax has been defined by the *Model Driven Architecture* (*MDA*) initiative of the *Object Management Group* (*OMG*).

execution platforms, and its architectural constraints (*e.g.*, concerning placement and collocation). Regarding the configuration protocol, particularly the distributed bindings configuration and the activation order of components that are the core of the present paper, [10] does not work in a decentralized fashion, and this harms the scalability of applications that can be deployed. Moreover, this works does not consider the reliability of the proposed protocol, whereas we focused here on the self-configuration verification and showed its necessity to detect subtle bugs.

[25] suggests an extension of *SmartFrog* [16] that enables an automated and optimized allocation of cloud resources for application deployment. It is based on a declarative description of the available resources and of the components building up a distributed application. Descriptions of applicative architectures and resources are defined using the *Distributed Application Description Language* (DADL). This language describes, on the one hand, the applications constraints related to the resources in terms of *Service Level Agreements* (*SLAs*) and, on the other hand, elasticity constraints. Compared to the present paper, [25] focuses on the language aspects and intends to address the optimal resources allocation. It does not give any details concerning the deployment process itself, which was our focus here.

There exist many approaches which aim at specifying and verifying distributed components and component-based architectures. In the 90s, several works [21,22,2,29] focused on dynamic reconfiguration of component-based systems, and proposed various formal notations (Darwin, Wright, etc.) to specify component-based systems whose architectures can evolve at runtime (addition/removal of components/bindings). Here, our goal was rather to verify the protocol at hand, to be sure that the corresponding Java implementation worked as expected. In [22,23], the authors show how to formally analyse behavioural models of components using LTSA. Another related work is [11], where the authors verify some temporal properties using model checking techniques on a dynamic reconfiguration protocol used in agent-based applications. Recently, [6] reported on the co-design and specification of the reconfiguration protocol of a component-based platform, intended as the foundation for building robust dynamic systems. The formal analysis of this protocol helped to detect several issues which were corrected in the corresponding implementation. CADP is richer in terms of verification techniques than LTSA, which does not propose any tool for equivalence checking for instance. Moreover, LNT user-friendliness and expressiveness for specifying both behaviours and data types (*e.g.*, FIFO buffers) makes it very convenient compared to other specification languages. Other toolboxes might have been used, such as SPIN [18] or MCRL2 [17]. LNT is more intuitive than Promela or the MCRL2 input language, and CADP also provides efficient model checking tools.

In [20], the authors present the formal verification of an operating system microkernel. They proved the functional correctness of the microkernel using the Isabelle theorem prover. The formal specification was generated automatically from an Haskell prototype, and the final implementation was manually encoded in C. This formal process helped to detect and correct many bugs in the system algorithms. Here, we focused on an alternative approach which requires much less effort in the verification process (automated versus semi-automated verification). Nevertheless, although model checking techniques are very suitable to detect bugs in any kind of application, they do not ensure correctness of the system as it may be achieved using theorem proving techniques.

In [3], the authors present a formal framework for behavioural specification of distributed Fractal components. This specification relies on the *pNet* model that serves as a low-level semantic framework for expressing the behaviour of various classes of distributed languages. They also propose a connection to CADP tools in order to check properties on these specifications. A graphical toolset for verifying AADL models is presented in [7]. This platform integrates several existing tools such as the NuSMV symbolic model checker or the MRMC probabilistic model checker. As far as autonomic systems are concerned, a few recent solutions have been proposed to analyse such systems. For example, in [28], the authors present the application of ASSL (Autonomic System Specification Language) to the NASA Voyager mission. In their paper, they show how liveness properties can be checked on ASSL specifications, and also plan to consider safety properties. The verification toolbox we use here already provides model checking techniques for liveness and safety properties, and many other formal analysis tools.

A preliminary version of this work has been presented in [27]. It is extended here in several aspects:

- the presentation of the protocol was extended, particularly with details on its implementation;
- the specification of the self-configuration protocol is presented with more details. In particular, we show and comment on several excerpts of the LNT specification;
- the dataset of applications consists now of 150 applications (100 before);
- we have added a subsection in Section 4 dedicated to experimental results where we present LTS sizes and computation times for several representative examples of our dataset;
- the related work section was revised and enhanced;
- we present in the conclusion some lessons learned from our experience.

6 Concluding Remarks

We have presented in this paper a cloud computing protocol self-configuring a set of components distributed over several VMs. This protocol is highly parallel and loosely-coupled, and this makes its design error-prone. In order to check that some key-properties are ensured, we have specified and verified it using state-of-the-art specification languages and verification tools. During the verification phase, we found a bug in the protocol using model checking techniques, which was corrected in the Java implementation.

As a result, we would like to emphasize lessons we have learned during this experience:

- Specification and verification techniques were introduced lately in the design process; the Java implementation was already available, but still under development. The goal was to detect bugs in pathological cases as the one we found. We could have started from the formal specification as advocated by classic software development processes, but this does not seem a good option for protocol designers who are not experts in formal methods. Coming up with code generation techniques might be an argument for convincing them to do so in the future.
- LNT, thanks to its user-friendly and programming-like notation, makes the formal specification accessible to non-experts and deeply simplifies the specification writing. Its expressive language enables the specification of concurrent behaviours and complex data types. In particular, LNT turned out to be suitable for specifying self-management protocols that exist in the lastest generation of component-based autonomic systems.
- The use of formal verification tools was successful because it helped to debug the protocol. All verification steps are fully automated, but the writing of temporal properties. However, we had to face state space explosion and this obliges us to validate applications involving only a few VMs.
- Formal techniques were used not only to chase bugs but also as a workbench for experimenting with different communication features (point-to-point, broadcast, different ways of implementing buffers, etc). This last point can particularly be of interest for optimizing an implementation (*e.g.*, the number of buffers) while preserving the same behaviour (*wrt.* a bisimulation notion [26] for example).
- This work shows that formal techniques and tools are not only of interest for critical systems but are also necessary for the design and development of complex system protocols existing in dynamically (re)configurable component-based systems.

A short-term perspective is to extend the protocol to take component failures into account. When a component fails, it may impact the whole application, yet we want our protocol to keep on starting and configuring as many VMs and components as possible. The extended protocol will be extensively validated using verification tools to check some new properties raised by the introduction of failure, *e.g.*, a component connected through a mandatory client interface to a failed component will never be started.

A long-term perspective would be to propose code generation techniques in OO programming languages (which is the main paradigm used in this community) for rapid prototyping purposes. This could also be used for implementing protocols starting from the formal specification and then generating code automatically.

Acknowledgements. The authors would like to thank Frédéric Lang for his help on compositional verification techniques and Wendelin Serwe for his comments on a former version of this paper. This work has been supported by the OpenCloudware French FSN project (2012-2014).

References

1. Open Virtualization Format Specification. Specification 1.1.0, Distributed Management Task Force DMTF Standard (2010)
2. Allen, R.B., Douence, R., Garlan, D.: Specifying and Analyzing Dynamic Software Architectures. In: Astesiano, E. (ed.) ETAPS 1998 and FASE 1998. LNCS, vol. 1382, pp. 21–37. Springer, Heidelberg (1998)
3. Barros, T., Ameur-Boulifa, R., Cansado, A., Henrio, L., Madelaine, E.: Behavioural Models for Distributed Fractal Components. Annales des Télécommunications 64(1-2), 25–43 (2009)
4. Bellissard, L., De Palma, N., Freyssinet, A., Herrmann, M., Lacourte, S.: An Agent Platform for Reliable Asynchronous Distributed Programming. In: Proc. of SRDS 1999, pp. 294–295. IEEE Computer Society (1999)
5. Bergamini, D., Descoubes, N., Joubert, C., Mateescu, R.: BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 581–585. Springer, Heidelberg (2005)
6. Boyer, F., Gruber, O., Salaün, G.: Specifying and Verifying the SYNERGY Reconfiguration Protocol with LOTOS NT and CADP. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 103–117. Springer, Heidelberg (2011)
7. Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V.Y., Noll, T., Roveri, M., Wimmer, R.: A Model Checker for AADL. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 562–565. Springer, Heidelberg (2010)
8. Broto, L., Hagimont, D., Stolf, P., De Palma, N., Temate, S.: Autonomic Management Policy Specification in Tune. In: Proc. of SAC 2008, pp. 1658–1663. ACM (2008)
9. Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., Powazny, V., Lang, F., Serwe, W., Smeding, G.: Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.4). INRIA/VASY (2011)
10. Chapman, C., Emmerich, W., Galán Márquez, F., Clayman, S., Galis, A.: Software Architecture Definition for On-demand Cloud Provisioning. In: Proc. of HPDC 2010, pp. 61–72. ACM Press (2010)
11. Cornejo, M.A., Garavel, H., Mateescu, R., De Palma, N.: Specification and Verification of a Dynamic Reconfiguration Protocol for Agent-Based Applications. In: Proc. of DAIS 2001. IFIP Conference Proceedings, vol. 198, pp. 229–244. Kluwer (2001)
12. Dillenseger, B.: CLIF, a Framework based on Fractal for Flexible, Distributed Load Testing. Annales des Télécommunications 64(1-2), 101–120 (2009)
13. Etchevers, X., Coupaye, T., Boyer, F., de Palma, N.: Self-Configuration of Distributed Applications in the Cloud. In: Proc. of CLOUD 2011, pp. 668–675. IEEE Computer Society (2011)
14. Etchevers, X., Coupaye, T., Boyer, F., de Palma, N., Salaün, G.: Automated Configuration of Legacy Applications in the Cloud. In: Proc. of UCC 2011, pp. 170–177. IEEE Computer Society (2011)

15. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
16. Goldsack, P., Guijarro, J., Loughran, S., Coles, A., Farrell, A., Lain, A., Murray, P., Toft, P.: The SmartFrog Configuration Management Framework. SIGOPS Oper. Syst. Rev. 43(1), 16–25 (2009)
17. Groote, J.F., Mathijssen, A., Reniers, M.A., Usenko, Y.S., Van Weerdenburg, M.: The Formal Specification Language mCRL2. In: Dagstuhl Seminars (2007)
18. Holzmann, G.J.: The Model Checker SPIN. IEEE Trans. Software Eng. 23(5), 279–295 (1997)
19. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization — Information Technology (2001)
20. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal Verification of an OS Kernel. In: Proc. of SOSP 2009, pp. 207–220. ACM Press (2009)
21. Kramer, J., Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management. IEEE TSE 16(11), 1293–1306 (1990)
22. Kramer, J., Magee, J.: Analysing Dynamic Change in Distributed Software Architectures. IEE Proceedings - Software 145(5), 146–154 (1998)
23. Magee, J., Kramer, J., Giannakopoulou, D.: Behaviour Analysis of Software Architectures. In: Proc. of WICSA 1999. IFIP Conference Proceedings, vol. 140, pp. 35–50. Kluwer (1999)
24. Mateescu, R., Thivolle, D.: A Model Checking Language for Concurrent Value-Passing Systems. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 148–164. Springer, Heidelberg (2008)
25. Mirkovic, J., Faber, T., Hsieh, P., Malayandisamu, G., Malavia, R.: DADL: Distributed Application Description Language. USC/ISI Technical Report ISI-TR-664 (2010)
26. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
27. Salaün, G., Etchevers, X., De Palma, N., Boyer, F., Coupaye, T.: Verification of a Self-configuration Protocol for Distributed Applications in the Cloud. In: Proc. of SAC 2012, pp. 1278–1283. ACM Press (2012)
28. Vassev, E., Hinchey, M., Quigley, A.: Model Checking for Autonomic Systems Specified with ASSL. In: Proc. of NFM 2009, pp. 16–25 (2009)
29. Wermelinger, M., Lopes, A., Fiadeiro, J.L.: A Graph Based Architectural (Re)configuration Language. In: Proc. of ESEC / SIGSOFT FSE 2001, pp. 21–32. ACM Press (2001)

Formal Modeling and Verification of Self-* Systems Based on Observer/Controller-Architectures

Florian Nafz, Jan-Philipp Steghöfer, Hella Seebach, and Wolfgang Reif

Institute for Software & Systems Engineering,
University of Augsburg, 86135 Augsburg, Germany
{nafz,steghoefer,seebach,reif}@informatik.uni-augsburg.de

Abstract. Self-* systems have the ability to adapt to a changing environment and to compensate component failures by reorganizing themselves. However, as these systems make autonomous decisions, their behavior is hard to predict. Without behavioral guarantees their acceptance, especially in safety critical applications, is arguable. This chapter presents a rigorous specification and verification approach for self-* systems that allows giving behavioral guarantees despite of the unpredictability of self-* properties. It is based on the Restore Invariant Approach that allows the developer to define a corridor of correct behavior in which the system shows the expected properties.

The approach defines relies (behavior the components can expect) and guarantees (behavior that each component will provide) to specify the general requirements on the interaction between the components of the system on a formal basis. If heterogeneous multi-agent systems with self-* properties are modeled so that relies are implied by the other components' guarantees, it is possible to formally verify correct system behavior. When using observer/controller architectures the approach also allows systematic decomposition and modular verification. We illustrate the approach by applying it to two different case studies – an adaptive production cell and autonomous virtual power plants.

Keywords: Adaptive Systems, Self-* Properties, Formal Methods, Verification, Multi-Agent Systems, Observer/Controller.

1 From Design Time to Runtime

Adaptive systems are not yet the silver bullet they are often hyped to be. It turns out that attempts to manage the complexity of modern cyber-physical systems or large-scale IT-systems often introduce a lot of complexity. While this might make the surrounding infrastructure simpler, e.g., by decreasing the number of administrators required to supervise a server farm, the transparency and controllability of the systems are reduced and thus is their trustworthiness. It is obvious that systems in which decisions are willingly relegated from design time to runtime pose many new challenges to regulators, standardization committees, and certification authorities, especially with regard to deployment of self-* systems in safety- and mission-critical domains.

One of the main tools in the certification of safety-critical systems are formal methods. A thorough formal analysis of a computer system can reveal flaws and bugs that are not identifiable by validation techniques such as testing. However, formal methods usually rely on sophisticated models of the system and its individual components. If the system is open and not all components are known at design time, it is impossible to create such a comprehensive model. Additionally, it is quite difficult to grasp the complex and diverse interactions that can occur in an open, adaptive system and it is even more difficult to verify all possible cases of interleaved communications.

If, however, it were possible to specify the external behavior of each system component in an abstract fashion and show that the individual components do not interfere with each other, internal models could be discarded while the behavior of the ensemble could still be verified. The rely/guarantee (R/G) paradigm first introduced by Jones in [21] and Misra and Chandy in [26] provides such a theoretical framework. It allows specifying guarantees provided by the components if they can rely on properties guaranteed by the environment or other components. This allows integrating arbitrary components without knowledge of their internal behavior, a major difference to most of the related approaches that are outlined in Sect. 2. The R/G paradigm is also ideally suited to capture the modularity of a system that can be decomposed into several types of components. We use this ability to decompose the system into a functional part and a part incorporating the self-* intelligence, represented by an observer/controller (o/c) [34]. This observer/controller architectural pattern encapsulates a feedback loop and is similar to the MAPE cycle [20] in the field of Autonomic Computing [28].

This chapter presents an integrated approach that enables the engineer to verify functional properties and thus give behavioral guarantees during design time without restricting the flexibility of the system during runtime. Its strengths are modularity, a top-down view of the system consistent with software engineering processes [38], and its independence of the self-adaptation algorithm used. It can therefore deal with arbitrary system changes at runtime and is scalable with respect to the number of agents in the system. The approach consists of the following elements:

- the Restore Invariant Approach (RIA), introduced in Sect. 3, is the theoretical framework required to model adaptive behavior and detect misbehavior at runtime;
- a verification approach based on RIA and the observer/controller architectural pattern (Sect. 4) that uses the rely/guarantee paradigm to show correct functional and reconfiguration behavior at design time as detailed in Sect. 5;
- an online result checking technique that allows the use of arbitrary self-* algorithms while maintaining functional correctness of the system, detailed in Sect. 6.

The target systems of this approach are systems based on an observer/controller architecture which implement their self-* properties by changing and adapting component configurations. In the following, we will always refer to self-* systems and imply this characteristic. This chapter focuses on the conceptional and theoretical foundations and omits the implementation details due to space restrictions, although the presented concepts were implemented for the case studies of an adaptive production scenario presented in Sect. 7 and autonomous virtual power plants, presented in Sect. 8. For more details on the implementation issues refer to [2, 30, 39].

2 State of the Art

There are several approaches for formal specification and verification of self-* systems related to the work presented here. This section presents the most important of them. Additional related work which focuses on single aspects of the overall approach will be introduced in the respective sections.

Wooldridge and Dunne state in [47] that the environment is essential for the verification of agents. They present a formal model in which the behavior of an agent and its interaction with the environment are described as a sequence of interleaved agent and environment actions. The framework used in this chapter reflects this idea and allows detailed modeling of the feedback loops in a self-* system, while still providing the ability for arbitrary system behavior. In contrast to Wooldridge and Dunne, the distinction of environment and system transitions in our approach is part of the used logical framework and thus allows the use of a comprehensive verification theory, including compositional reasoning with rely/guarantee. Further the behavior is restricted by a corridor of correct behavior formulated by constraints, which allows to specify the agent's behavior on an abstract level without having to consider the particular implementation.

In [41], Smith and Sanders present a top-down approach for incremental formal development of self-organizing systems. An abstract specification for the complete system is refined stepwise down to component level. The correctness of the system is ensured by verification of the refinement steps. The Z notation is used as specification formalism. Their approach does not distinguish self-* and functional behavior, as they do not focus on a particular architecture. Instead they look at various applications and show how refinement can be applied in each specific case. In contrast, by focusing on an observer/controller-architecture, we can derive generic properties for applications based on this architecture. Nevertheless, their work provides good strategies for the refinement between different abstraction layers which are similar to the decomposition steps presented in this chapter and can provide useful guidance for further refinement on agent level, e.g., when considering hierarchical observer/controller-architectures.

Giese et al. present a modeling and verification approach in [9, 18] for self-adaptive mechatronic systems. The interaction between the components is modeled by so called coordination patterns describing the structural adaptation process. The system states are modeled as graphs and their dynamic behavior as graph transformations. A compositional verification approach also utilizing the rely/guarantee paradigm allows verifying safety properties that can be formalized as structural invariants over the graph transformation system. The coordination patterns in their approach are similar to parts of the corridor specification in the Restore Invariant Approach, as both are specifying a correct system structure. The rely/guarantees used here specify the requirements on the components' behavior in order to exhibit the desired properties as long the system is within the corridor and the requirements on the self-* process. Their approach does not make this distinction and directly uses the specification of the component behavior, which together with the coordination patterns combines functional and self-* behavior. Their approach therefore does not allow to change the implementation of the self-* behavior without the need of performing the complete verification again.

There are a number of further approaches focused on analyzing adaptive systems. Kramer and Magee [24] use automata to specify the properties of an adaptive system and use LTSA (Labelled Transition System Analyser) for automatic analysis of execution scenarios. Their approach does not consider modular reasoning as presented here. In [49] Zhang et al. present a modular approach based on model checking for adaptive programs against global invariants and transitional properties formulated in Linear Temporal Logic (LTL). They present a model checking algorithm for the verification of adaptive programs, such as an adaptive routing protocol. In contrast to the work here, they focus solely on the verification technique and not on the specification of adaptive systems. They also do not consider the formalization of uncertainty introduced by the environment.

3 The Restore Invariant Approach

The Restore Invariant Approach (RIA) allows defining a corridor of correct behavior. The system tries to operate within the corridor as long as possible. Due to unexpected disturbances the system leaves the corridor. Disturbances can be changes in the environment, failures, new or leaving agents, or new objectives, for instance. Whenever the corridor is left the system initiates a self-* phase and tries to reconfigure in order to return to the corridor. This concept is further elaborated in Sect. 3.1. As Sect. 3.2 shows, the concepts of RIA and the behavioral corridor are the foundation for system verification. An additional element necessary to enable successful reconfiguration is a safe state the system can reach in case of a failure, as outlined in Sect. 3.3. Finally, the concepts of RIA also allow to give a clear distinction between systems that have self-* properties and those that don't as described in Sect. 3.4.

3.1 Corridors of Correct Behavior

The basic idea behind the Restore Invariant Approach is to constrain the behavior of the system so that it only exhibits correct behavior. An advantage of this approach is that the system retains its flexibility and is still able to adapt during runtime and make decisions autonomously.

From a formal point of view, a system can be described as a transition system $SYS = (S, \rightarrow, I, AP, L)$, where S is the set of states, $\rightarrow \subseteq S \times S$ a transition relation, $I \subseteq S$ a set of initial states, AP a set of atomic propositions and L a labeling function. A trace π of the system is then given by a sequence of states $s_i \in S$ whose states are related by the transition relation and which starts in an initial state s_0.

$$\pi = s_0, s_1, s_2, \dots, s_n$$

Fig. 1 shows an example trace of an abstract transition system SYS which tries to stay within the corridor. The system recognizes a violation of the corridor and triggers a self-* process in order to reach a state within the corridor.

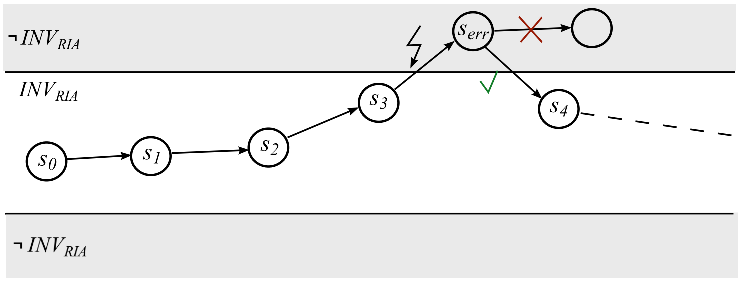


Fig. 1. Corridor of correct behavior of a self-* system [31]

Formally the specification of the corridor corresponds to a predicate logic formula[1] – the invariant INV_{RIA} – which is evaluated over a system state. The term "invariant" is used as the system's goal is to maintain the invariant on the entire system trace. The invariant differentiates the system states into those that exhibit correct behavior and those that do not. This allows to separate the states into two disjoint sets: a set S_{func} of functional states and a set S_{reconf} of reconfiguration states. The functional states are states within the corridor in which the system shows its desired behavior. The reconfiguration states are states outside the corridor in which reconfiguration is necessary. This abstract definition can accommodate a variety of situations in the system that can lead to adaptations. If, e.g., new agents that enter the system should trigger a reconfiguration, the invariant will have to be formulated so that an idle agent or one that has not been configured violates it. The system will then switch to a reconfiguration state as soon as such a situation occurs.

Related Work: In [17], Gärtner presents a similar classification of the state space for fault tolerant systems. He distinguishes three kinds of states: a set of *invariant states*, in which the system exhibits the desired properties, corresponding to the functional states of RIA; a set of states constituting the *fault span*, containing all invariant states and additionally all states which are tolerable by the system and from which the system eventually returns into an invariant state; finally, the set of all possible states.

Another classification of the state space of Organic Computing systems is proposed by Schmeck et al. in [37]. The *target space* contains the states the system should try to reach. If this is not possible, the system should at least try to get into a state of the *acceptance space*. The *survival space* consists of all states outside the acceptance space from which the system can get back into the acceptance or target space. All remaining

[1] Theoretically, a temporal logic formula could be used instead of a predicate logic formula. However, it is unclear how a system can evaluate a temporal invariant during runtime and decide whether it is violated or not. In order to decide this, the system would have to predict the future behavior. In the area of runtime verification the correctness of temporal logic properties is checked during runtime. For example, Leucker et al. try to monitor temporal logic properties during runtime [6]. In each step the property can be true, false or inconclusive. In this chapter predicate logic is used to formulate the invariant, although the general approach is not limited to it. The use of predicate logic means that the invariant can be evaluated in each state and it can be decided whether a state is within the corridor or not.

states are states within the *dead space* with no possibility to get back into the acceptance space. Compared to the corridors of RIA, Schmeck et al. split the functional states into target and acceptance space to distinguish optimal and non-optimal but correct states.

Both classifications separate the reconfiguration states into a set of states in which a path back into a functional state exists and a set where no path exists anymore. The classifications are used to describe the behavior of a self-* system on an abstract level. The specification of behavioral corridors in RIA exceeds these classifications by providing the tools to clearly define the different sets of states and to use these definitions both at design time to provide techniques for formal analysis (see Sect. 5) as well as at runtime to monitor the correct behavior of the system (see Sect. 6).

3.2 Behavioral Guarantees Based on RIA

By distinguishing functional and reconfiguration states, the requirements for the self-* properties of the system can be specified using the invariant. Whenever the invariant is violated, the system has to try to return to the corridor and to restore the invariant. The invariant is also a sufficient condition for system states that exhibit the expected behavior. That means that the system exhibits correct behavior when in a state in which the invariant holds. The correctness of the functional behavior of the system can therefore be verified independently of the self-* mechanisms. For the verification of the functional system it is assumed that there exists a mechanism that restores the invariant when it is violated. For a specific self-* mechanism it has to be proven that this assumption holds.

The definition of corridors has several more advantages compared to an explicit listing of all states. First, it is usually hard or expensive to find and list all states that are valid. It is often easier to formulate common properties that valid states need to exhibit. The abstraction induced by the invariant reduces the complexity of formal reasoning and the separate treatment of functional properties and self-* behavior can be exploited in order to give behavioral guarantees. In Sect. 5, a more detailed insight into this and an approach for providing behavioral guarantees will be given.

3.3 Safe Reconfiguration with Quiescent States

To ensure correct reconfiguration the system may not perform any actions that interfere with the reconfiguration process. Therefore the first task in a self-* phase is to transition the involved system components to a consistent and passive state in which they perform no critical actions. In literature this state is often called *quiescent state* [13, 22, 33, 48]. Kramer and Magee define a quiescent state in [22] as a state in which an agent is in a locally consistent and passive state, where it performs no actions which disrupt the reconfiguration. They also identify a quiescent state as a necessary condition for reconfiguration [23]. In a later work Vandewoude et al. [45] presented *tranquility*, a weaker condition for consistent reconfiguration. It allows an agent to still be involved in a transaction if it stops actively processing requests. As quiescence implies tranquility we use the stronger concept in the following, although it is possible to use the weaker condition in order to specify the requirements on an agent's behavior regarding a consistent reconfiguration.

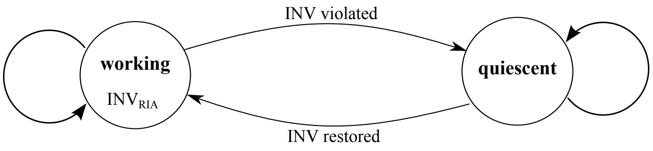


Fig. 2. Abtract view of a system's behavior

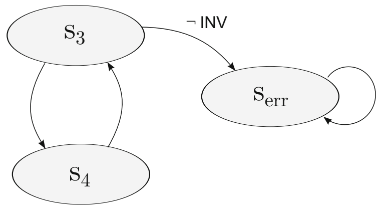
Fig. 2 shows the life-cycle of a self-* system that uses the Restore Invariant Approach. As long as the invariant holds, the system is within the corridor and can exhibit the expected behavior. When a failure occurs, the invariant is violated and the system starts a self-* phase. The first step is a transition to a quiescent state to be able to perform the actual reconfiguration process. In many cases, it is not necessary for the whole system to enter a quiescent state. Often it is sufficient that only the affected part of the system becomes quiescent, while the rest of the system can still be working. A challenge here is to identify the parts that need to be included into the reconfiguration. This question is not examined here. For details on this topic refer to [1,40]. As soon as reconfiguration is finished and the invariant is restored, the system leaves the quiescent state and starts working again (functional phase).

What quiescence means is application-specific and has to be defined in the context of the considered application. A quiescent state can be a truly passive state in which an agent stops all actions until the reconfiguration process finishes (see Sect. 7) but also a state in which the agent continues acting according to the old configuration until the new one is calculated (see Sect. 8). The specifics of the quiescent state depend on the kind of reconfiguration used in a system and the conditions for a consistent switch to a new configuration. In Sect. 5 we will have a closer look at how these transitions are initiated with respect to a certain system architecture.

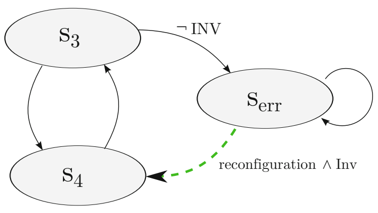
3.4 Comparing Systems with and without Self-* Properties

Based on the classification of functional and reconfiguration states, the difference between systems with and without self-* properties can be explained. Fig. 3 shows an abstract system with three states. If the system leaves the corridor and enters a state s_{err} that violates the invariant, it is therefore outside the corridor. In safety-critical applications, s_{err} is typically some kind of fail-safe state to avoid harm to human beings and the system's environment. In a fail-safe state the system still fulfills its safety properties, but usually does not guarantee any liveness properties such as progress or termination [17].

A traditional system with no self-* capabilities nor redundancy cannot reach a functional state once it has reached an error state (see Fig. 3(a)). In contrast, a self-* system (Fig. 3(b)) can return to a functional state (s_4), e.g., by reconfiguring itself. This implies that the relevant agents have to be put into a quiescent state in order to be reconfiguered consistently. During the reconfiguration, the system is changed so that redundancy within the system can be used to compensate for failures. Thus, component failures reduce the level of redundancy limiting potential future reconfigurations. The changes in the system are often enacted as part of a self-* process that changes the internal structure of the system. We distinguish two kinds of redundancy:



(a) System without self-* capabilities



(b) System with self-* capabilities

Fig. 3. Behavior of a traditional system compared to a system with self-* capabilites

local redundancy, where one part of the system contains all redundancy; and distributed redundancy, where the redundancy is spread over the system. While the former can be exploited without the need of self-* properties, the latter can only be enabled by enacting a new system structure. Such distributed redundancy plus self-* properties is also able to compensate for the complete failure of a part of the system. Pure local redundancy is always subject to single-point failures and thus limits a system's recoverability. Hence the combination of redundancy and self-* properties can considerably contribute to a robust and flexible system.

4 Observer/Controller-Architectures

So far, the self-* system was considered in a very abstract and formal fashion. In this section, the observer/controller-architecture, a common architecture of self-* systems is considered. Based on this architecture, the formal modeling and verification approach with RIA is explained in detail in Sect. 5.

A way to integrate adaptivity in a system is the introduction of feedback loops [12, 14]. A change in the system or its environment triggers a reaction within the system that causes a subsequent change and so forth. Such loops can be used to model the adaptiveness of a system and to understand the dynamics that occur in a system [42].

In this work, the generic observer/controller-architecture proposed by Richter et al. in [34] is used and refined. The architecture shown in Fig. 4 is one realization of the feedback loop principle: a functional system is observed and the observations are reported to the controller which in turn effects the system in a way that it deems best to reach the system's goals. The actual effect is monitored by the observer. Thus, a feedback loop is established that guides the adaptation of the system and its behavior.

The functional system in Fig. 4 consists of several autonomous, interacting components, so called agents. These components react to control signals from the controller component, but only in case of changes in the environment which necessitate a reconfiguration of the functional system. As long as the system behaves correctly according to the corridor, the o/c-layer is passive and does not interfere with the rest of the system. However, the observer monitors the functional system which is equivalent to the monitoring of the invariant. Whenever the invariant is violated, the observer notifies the controller. The controller then initiates a reconfiguration of the system. After it has advised the agent to enter a quiescent state, it starts a reconfiguration mechanism

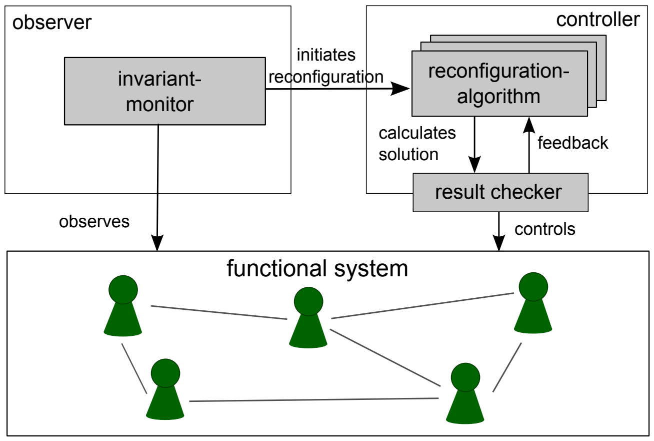


Fig. 4. An observer/controller-architecture for systems using RIA [16]. It contains a monitor component to observe the invariant as well as a result checker to verify solutions of the reconfiguration algorithm.

calculating a new configuration for the system. The new configuration has to fulfill the invariant and thus, the system will again exhibit correct behavior after the reconfiguration. The interaction between the o/c-layer and the functional system is always observer/controller-initiated.

Fig. 4 suggests that the o/c-layer is a central instance within the system. This is a sophism since the architecture can be realized in several ways [11]. Depending on the system's properties and its application area, each agent can have an individual o/c-layer, thus achieving a completely decentralized architecture. Multiple layers of observation and control can be implemented achieving a decentralized, hierarchical architecture.

Components in the system interact horizontally in each layer and vertically between the layers. Interactions can thus take place only between agents in the functional layer, between the o/c-layer and the functional layer, and between several o/c-layers, depending on the chosen architecture. Again, what all these possible variations of o/c-architectures have in common is the strict separation of functional system and o/c-layer which enables the specification and formal analysis of desired system properties as well as behavioral guarantees as shown in this chapter.

5 Formal Model of an O/C-Based System

In this section a generic formalization of self-* systems with an o/c-architecture is presented. Based on the formal framework described in Sect. 5.1 and a compositional reasoning paradigm outlined in Sect. 5.2, a formal model for systems based on o/c-architectures is developed in Sect. 5.3. After showing how such a system can be decomposed properly in Sect. 5.4, conditions for correct behavior of the individual components and interaction between the layers are formulated in Sect. 5.5. The formulated conditions can be instantiated for specific applications to retrieve the proof obligations for the particular agents and the o/c-interaction, effectively allowing a composition verification approach.

5.1 Formal Framework

To begin with, we want to give a short overview of the formal framework used for modeling and verification. The full details, including a specialized logic and calculus along with the respective semantics, can be found in [4, 8]. For a tool-supported verification, these elements have been integrated in the interactive theorem prover KIV [5].

From a formal point of view a run of the system is a sequence of states, which is called a trace. A state is defined by an evaluation of the systems variables V. A step consists of a so called *system transition* followed by an *environment transition*. A step therefore consists of three states: an unprimed state s_i at the beginning of the step, an intermediate primed state s_i' formalizing the evaluation after the system transition, and a double primed state s_i'' for the evaluation after the environment transition. The double primed state is equal to the unprimed state for the subsequent step ($s_i'' = s_{i+1}$). Thus the system and environment transition alternate, as depicted in Fig. 5. This trace based view of the system is necessary as the properties one expects from the system and the guarantees about its behavior are temporal properties. Desired guarantees are, e.g., that the system *never* shows some unwanted behavior or a property *always* holds.

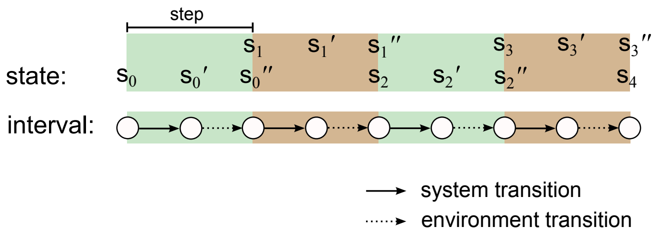


Fig. 5. Relation between unprimed, primed and double primed states. A step consists of a system transition followed by an environment transition.

Besides the variables v there are also primed and double primed variables in order to accurately formalize the transitions. For each variable $v \in V$ there is a corresponding primed variable v' and double primed variable v''. The sets of all primed/double-primed variables is denoted accordingly by V' and V''.

A *system transition* ($s_i \mapsto s_i'$) can therefore be formulated as a predicate logic formula over V and V' that describes the relation between the values of the variables $v \in V$ before and after the system transition. An *environment transition* ($s_i' \mapsto s_i''$) analogously describes the changes during an environment transition. As the double primed state is the unprimed state of the successive state, the value of v'' in state s_i'' is equal to the value of v in the next successive state s_{i+1}. For example, $x' = x + 1$ expresses that the system increases x by one and $x'' = x'$ expresses that during an environment transition x is not changed. If there is no statement made for a variable in the environment step, this means that the value can be changed arbitrarily by the environment. This is a crucial feature of the framework: if no restriction is put in place, no assumption is made of what the environment is capable of. It is therefore not necessary to explicitly model everything the environment can do, but verify the system for completely arbitrary changes in the environment by consciously under-specifying certain aspects.

While it is a great advantage to be able to leave many aspects of the environment open, an explicit model of *some* of the behavior of the environment can be very beneficial in the case of self-* systems [47]. If regarded properly, the system boundary can be clearly established and thus a clear separation between the system and its environment can be achieved. This also aids in the modeling of the interactions between the system and its environment, as exemplified in Sect. 7 and Sect. 8.

Parallel components are expressed through an interleaving operator $\|$. The interleaving of two components (agents) Ag_i and Ag_j means that either Ag_i or Ag_j can make a system transition. The particular agent cannot distinguish how many transitions the other agents have done between two of its steps. From its local point of view everything occurred in a single environment transition. That means from an agent's point of view the system transitions of the other agents are in its environment transitions as well as changes made by the global environment. This is illustrated in Fig. 6.

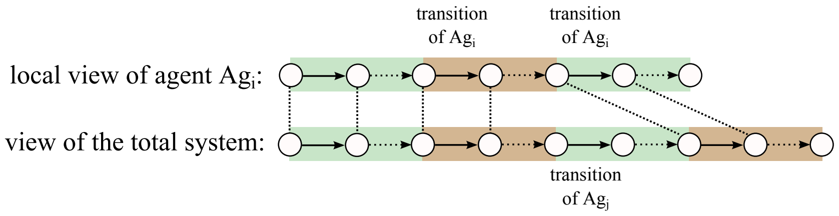


Fig. 6. Local view of an agent Ag_i and its relation to the run of the total system

5.2 Compositional Reasoning with Rely/Guarantee

So far we established a global view of the system. However, the goal is to retrieve properties of single agents and to have a local view on the system but still be able to guarantee properties of the complete system. The observer/controller-architecture provides a natural way for the decomposition into several subcomponents. This is depicted in Fig. 7. The complete system can at first be split into the observer/controller o/c and

$$\begin{tikzcd}
& \mathit{Self\text{-}*\ SYS} \arrow[dl] \arrow[dr] & & \\
o/c & \| & SYS_{func} \arrow[dl] \arrow[d] \arrow[dr] & \\
& Ag_1 \| & \cdots \| & Ag_n
\end{tikzcd}$$

Fig. 7. Compositional view of the system structure

the functional system SYS_{func}. Both are running in parallel which is represented by the interleaving operator $||$. The functional system can again be split into several agents $(Ag_1, \ldots, Ag_n)$ that are running in parallel as well. Of course, the observer/controller can also consist of several parallel components. This is not considered here, as for the verification of functional properties only the *specification* of the complete o/c-layer is required. However, the approach presented for the functional system works for a decomposition of the o/c as well.

This modular structure can be used for a compositional verification approach. The idea behind compositional verification is to reason about properties of the global system by proving properties of single components only. The main advantage is that reasoning over single components is usually less complex then reasoning over a parallel system. A common compositional proof technique is the *rely/guarantee paradigm* which is used here and was introduced by Jones in [21] and by Misra and Chandy under the term assumption-commitment in [26].

The basic idea is that each component guarantees a specific behavior as long as it can rely on some properties of its environment. The behavior of a component is specified by a *guarantee* $G(V, V')$ provided by the component. This is expressed as a predicate over the component's transitions. To be able to guarantee the specified behavior, the component needs to be able to make assumptions about its environment, as it *relies* on certain – but not necessarily completely specified – aspects of behavior of its environment. If no relies are formulated at all and the environment is thus completely arbitrary, a component will not be able to give any guarantees, as every system action can immediately be revoked by the environment[2]. To create relies that limit the behavior of the environment as little as possible, they are usually defined by excluding some particular behavior. A typical property of the environment is that it does not change a component's internal variables. Formally, a rely $R(V', V'')$ is specified over the environment transitions.

The behavior of a component Ag_i can then be specified using both rely and guarantee. As long as the rely $R_i(V', V'')$ holds, the component guarantees $G_i(V, V')$. This property is formalized as $R_i(V', V'') \overset{+}{\rightarrow} G_i(V, V')$. The rely/guarantee specification abstracts from the internal implementation of the component and specifies the external behavior a component should exhibit. It is therefore a black box specification.

In order to be able to reason about the global system, a compositionality theorem developed by Bäumler et al. [7] is used. It describes the necessary correlations between the local rely/guarantees R_i/G_i of the components C_i and defines the proof obligations in order to guarantee a global rely/guarantee property R/G of the combined system. The main obligation is to prove that each component behaves according to its local rely/guarantee specification. The other obligations ensure the compatibility and consistency among the rely/guarantees, e.g., the guarantee of one component does not violate the rely of another component.

Theorem 1 (Compositionality theorem). *If:*

i. *for all* $i = 1, \ldots, n : C_i, Init(V) \vdash R_i(V', V'') \overset{+}{\rightarrow} G_i(V, V')$
ii. *for all* $i = 1, \ldots, n : G_i(V', V'') \rightarrow G(V', V'') \wedge \left(\bigwedge_{j=1,\ldots,n \wedge j \neq i} R_j(V', V'')\right)$
iii. *for all* $i = 1, \ldots, n : G_i(V, V)$

[2] Note, that from the local view of a component, the environment contains all other components.

iv. *for all* $i = 1, \ldots, n : R_i(V,V') \wedge R_i(V',V'') \rightarrow R_i(V,V'')$
v. $R(V',V'') \rightarrow \left(\bigwedge_{i=1,\ldots,n} R_i(V',V'')\right)$
vi. $\exists\, V \;:\; \mathit{Init}(V)$

then: $C_1 \,||\ldots||C_n, \mathit{Init}(V) \vdash R(V',V'') \overset{+}{\rightarrow} G(V,V')))$

The informal meaning of the proof obligations of this theorem are as follows:

i. All components must sustain their guarantee as long as the rely holds. It can be assumed that an initial condition $\mathit{Init}(V)$ holds in the first step.
ii. The guarantee of each component preserves the global guarantee and does not violate the relies of all other components.
iii. The local guarantee is reflexive, that means it must hold if nothing (no variable) is changed.
iv. The relies of all components are transitive. With this property, a component's relies are preserved even if other components make several steps in a row.
v. All component relies hold if the global rely holds. Therefore, no component rely is violated in the environment step. This implies that an agent cannot assume that no failures occur, for instance.
vi. An initial configuration for the system must exist. This ensures that obligation *i* is consistent.

If the rely/guarantees fulfill these properties and the component implementation is correct with respect to its local rely/guarantee property then the global guarantee holds for the complete interleaved system.

If the system consists of several identical components of the same type only one of these components has to be proven. The theorem then allows to reason about a system consisting of an arbitrary (but finite) number of components running in parallel, also including changing numbers of agents. The proofs are also valid when the number of agents changes during runtime. Thus, the kinds of adaptivity that can be covered with this technique include component failures, as well as adding and removing agents. The adaptivity of the components is realized by changing their parameters.

The compositionality theorem was proven with the interactive theorem prover KIV and can therefore be directly applied during a proof for a particular system. The advantage is that the reasoning is tool-supported and that for a particular system only the local R/G properties have to verified against the components implementation. More information about the theorem, the resulting proof obligations and technical details can be found in [36].

5.3 Formal Model of a System Based on an Observer/Controller-Architecture

With these formal foundations in place, it is now possible to define an abstract formal model for systems based on an observer/controller-architecture that can be used to formally specify the corridor of correct behavior and the requirements for a correct o/c implementation. The model can then be instantiated for a concrete system to provide behavioral guarantees by formal verification of the functional system part and an observer/controller implementation against their respective specifications.

The system can be described as a set of variables V_{all} which is split into a set V_{func} of variables defining the state of the functional system and a set V_{env} describing the system's environment. The variables in V_{func} again can be split into the following disjoint sets, as depicted in Fig. 8:

– A set V_{conf} of variables which contain the configuration of the functional system. These are the variables (parameters) that can be changed by the observer/controller during a self-* phase in order to restore the invariant.
– A set V_{int} of variables that contains the agent's internal variables. They can only be changed by the agents. This set contains variables which the agents use for internal calculations and to store intermediate data.
– All other variables are in V_{rest}. These variables can be changed by the environment and describe, e.g., sensor data or hardware status which can both include errors or be changed at random points in time.

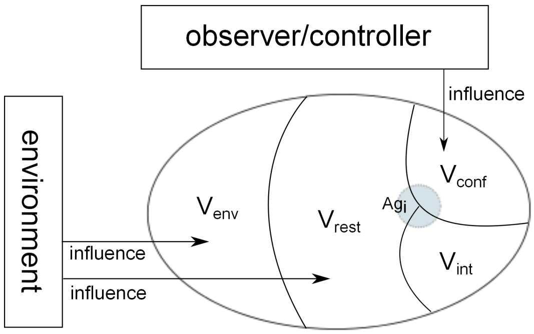


Fig. 8. Abstract o/c-system with different variable sets for environment and functional system

The set of the variables of the functional system therefore is defined as $V_{func} := V_{conf} \cup V_{int} \cup V_{rest}$. The set of V_{env} models the environment of the system and allows to express the effect an agent's action has on its environment. As the agents' actions alter the environment, and the environment and the agents are interleaved, feedback loops can thus be formalized.

Each agent has its own set of variables V_{func}. Additionally the agents have two variables (flags) *reconf* and *deficient* which model the o/c-interaction. The flag *reconf* signals an agent that a self-* phase has started and that it should behave passively. The flag *deficient* signals an agent that a reconfiguration occurred. The second flag is necessary as it is theoretically possible that between two steps of an agent several steps of the remaining system – including a complete reconfiguration – occurred due to the asynchronous execution. In a concrete implementation these flags not necessarily have to be flags that are set by the o/c directly, they can also be refined to a message passing communication model. Such a model allows interactions between the layers to be conveyed by messages and complex protocols, e.g., handshake protocols with multiple messages.

Formally the state of an agent is represented as a tuple:

$$\text{state}_{ag} := (\ .\text{id} : \text{ID} \times .\text{reconf: bool} \times .\text{deficient} : \text{bool} \\ \times .\text{vconf} : V_{conf} \times .\text{vint} : V_{int} \times .\text{vrest} : V_{rest});$$

The dynamics are modeled as state transitions specifying how the variables of an agent change during a system step. Formally, this is expressed as transition predicates relating unprimed and primed variables.

5.4 Decomposition of the Observer/Controller-Architecture

As described above and illustrated in Fig. 7, the self-* system can be decomposed into several components running in parallel. If we apply the rely/guarantee approach to the observer/controller-architecture the system can be decomposed in two steps (see Fig. 9).

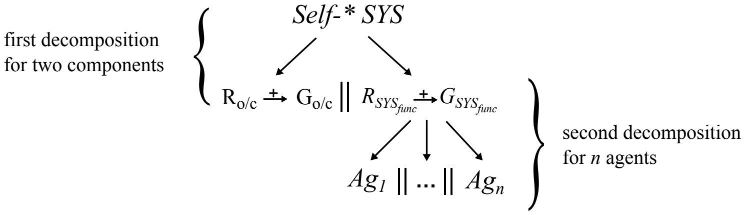

first decomposition for two components
Self-* SYS
$R_{o/c} \xrightarrow{+} G_{o/c} \parallel R_{SYS_{func}} \xrightarrow{+} G_{SYS_{func}}$
$Ag_1 \parallel ... \parallel Ag_n$
second decomposition for n agents

Fig. 9. System structure and compositionality

At first the system is decomposed into the o/c part and the functional system. The behavior of both is specified with corresponding rely/guarantee properties. On this level we have a parallel system consisting of two components. For a particular implementation, the following has to be proven:

– For the o/c part it must be proven that the implementation of the observer/controller (o/c_{impl}) is correct with respect to its specification.

$$o/c_{impl} \models R_{o/c} \xrightarrow{+} G_{o/c} \quad (1)$$

– For the functional system it must be verified that the agents behave according to their rely/guarantee property.

$$SYS_{func} \models R_{SYS_{func}} \xrightarrow{+} G_{SYS_{func}} \quad (2)$$

The first decomposition leads to a separation of concerns. Both system parts can be treated separately. The rely/guarantee properties specify the interaction between both layers. This specification and the properties each layer must guarantee ensure that the complete system exhibits the expected properties. The correctness of the global system is ensured by a compositionality theorem.

The functional system itself consists again of a number of agents and can be decomposed in a second step into several agents. The course of action is the same as for the first decomposition step. The rely/guarantee property of the functional system

(Eq. 2) is broken down to local rely/guarantee properties for the individual agents and are enriched with properties about the interaction between the agents themselves. For a particular implementation Ag_i of an agent it must be proven that it is correct according to its R/G specification:

$$Ag_i \models R_i \xrightarrow{+} G_i \tag{3}$$

The R/G specification contains the interaction of the agent with its observer/controller and other agents with shared variables. It also contains, e.g., the individual contribution of one agent to the global task.

In the next sub-section we define generic rely/guarantee properties for the first decomposition step. They specify the requirements on the interaction between the observer/controller and the functional system in order to prove that a property *Prop* always holds.

5.5 Rely-Guarantee Definition of the Interaction between Functional and Self-*-Layers

Before we consider the verification of Eq. 1 and Eq. 2 against a particular implementation, we need to specify the rely/guarantee properties first. The observer/controller-architecture reflects the distinction of the two phases of the restore invariant approach. The functional part is mainly responsible for establishing the functional properties of the system and therefore is active during the functional phases. The observer/controller-layer is responsible for monitoring the invariant and reconfiguration in case of an invariant violation. It puts the functional part into a quiescent state and is mainly active during the self-* phases.

Observer/Controller Specification. First, we look at the observer/controller specification and its relies and guarantees. A correct o/c implementation has to guarantee that at the end of every self-* phase the invariant is restored. That means whenever the o/c signals an agent to leave its quiescent state, INV_{RIA} must hold.

Note that this does not require that the o/c always finds a solution. This would imply a perfect o/c, which is not realistic as sometimes there is no solution possible, e.g., when no more redundancy is available in the system. Eq. 4 is sufficient to guarantee safety properties. Additional requirements can be added to express some quality criteria concerning the reconfiguration. Further, the observer/controller has to guarantee that it does not interfere with the functional system in functional phases and that it always puts the agent into a quiescent state before changing the configuration parameters ($noInterference(V_{all}, V'_{all})$). It also guarantees not to change the agent's internal variables ($Unchg_{sys}(V_{func} \setminus V_{conf})$) and not to violate the system properties that should be proven (denoted here by *Prop*). These system properties are usually defined by the developer and are retrieved from the requirements on a particular system.

$$\begin{aligned} G_{o/c}(V_{all}, V'_{all}) :\leftrightarrow \quad & (\forall\, i\, :\, (Ag_i.reconf \land \neg\, Ag'_i.reconf \rightarrow INV_{RIA}(V'_{all}))) \\ & \land (\forall\, i\, :\, \neg\, Ag'_i.reconf \rightarrow noInterference(V_{all}, V'_{all})) \\ & \land (Prop(V_{all}) \rightarrow Prop(V'_{all})) \\ & \land Unchg_{sys}(V_{func} \setminus V_{conf}) \end{aligned} \tag{4}$$

To be able to guarantee this behavior the observer/controller relies on the agents not to leave their quiescent state themselves. In terms of the reconf variable, this means that it is only changed on the o/c's initiative.

$$\begin{aligned} R_{o/c}(V'_{all}, V''_{all}) :\leftrightarrow\ & (\forall\, i\ :\ Ag'_i.reconf \leftrightarrow Ag''_i.reconf) \\ & \wedge (Prop(V'_{all}) \rightarrow Prop(V''_{all})) \end{aligned}$$

The observer/controller also assumes that the property is maintained by the rest of the system (all the agents currently participating in the system). This is necessary as we want to prove that the property is never violated by the complete system and the functional system is part of the system as well.

Functional System Specification. The functional system guarantees that it does not change the configuration on its own. It also has to guarantee the considered property *Prop*. Further, the functional system must guarantee to be quiescent during the self-* phase and only to leave the quiescent state on o/c notifications.

$$\begin{aligned} G_{SYS_{func}}(V_{all}, V'_{all}) :\leftrightarrow\ & V_{conf} = V'_{conf} \\ & \wedge (Prop(V_{all}) \rightarrow Prop(V'_{all})) \\ & \wedge (\forall\, i\ :\ Ag_i.reconf \rightarrow quiescence(V_{all}, V'_{all})) \\ & \wedge (\forall\, i\ :\ Ag_i.reconf \rightarrow Ag'_i.reconf) \end{aligned}$$

To ensure this, it relies on a correct o/c behavior, i.e., the o/c only changes the configuration variables in self-* phases and the internal variables of the agents are only changed by themselves. It further relies on the remaining part of the system not to violate the expected property as well.

$$\begin{aligned} R_{SYS_{func}}(V'_{all}, V''_{all}) :\leftrightarrow\ & (\forall\, i\ :\ \neg Ag''_i.reconf \wedge \neg Ag''_i.deficient \rightarrow Unchg_{env}(V_{conf})) \\ & \wedge (\forall\, i\ :\ Ag'_i.deficient \rightarrow Ag''_i.deficient) \\ & \wedge (Prop(V_{all}) \rightarrow Prop(V'_{all})) \\ & \wedge Unchg_{env}(V_{int}) \end{aligned}$$

Applying the compositionality theorem from Sect. 5.2 it can be proven that the abstract property *Prop* also holds for the combined system if both parts behave according to their rely/guarantee specification. The environment of the complete system is still allowed to arbitrarily change the environment (V_{env}) and the agents' variables V_{rest} which are, e.g., specifying the agents' sensor data or hardware status. It is just assumed that the environment cannot change an agent's internal variables or configuration parameters.

5.6 The Verification Process

We now have all elements in place that are required to actually verify a concrete application. This process is exemplified with two case studies in Sect. 7 and Sect. 8. In all cases, four general steps have to be followed:

1. Define formal model, system dynamics, and property:
   - define a formal model of the functional system, respectively the agents;
   - define the variables describing the system state;
   - specify the dynamic behavior, i.e. the state changes;
   - specify the property the functional system has to adhere to.
2. Define the corridor and reconfiguration behavior:
   - specify the application specific corridor based on the formal model;
   - define what the quiescent state of an agent is;
   - specify what *noInterference* means for the observer/controller in the particular application.
3. Instantiate the abstract rely/guarantees in this section with the application specific variables and formulas:
   - for the observer/controller;
   - for the functional system.
4. Verify that the observer/controller and the functional system behave according to the instantiated rely/guarantee properties.

Following this course of action will verify that the functional system behaves according to its specification and that the interaction between observer/controller and the functional system is correct. This includes all behavior that takes place when the system is reconfigured. However, the actual reconfiguration algorithm implemented in the controller part of the o/c is not subject to verification. Therefore, an additional measure has to be put in place. This measure is discussed in the following section before the verification process is exemplified with two case studies.

6 Observer/Controller Correctness by Verified Result Checking

The verification of the functional system relies on a correct observer/controller-layer. For the verification of the functional system it was assumed that the observer/controller applies configurations that fulfill the invariant. To verify this property, one option is to reason about the reconfiguration algorithm in question by direct verification. Depending on the algorithms' complexity this task can be arbitrarily difficult or even infeasible as often bio-inspired algorithms, learning techniques or stochastic approaches are used to implement the self-* features. These algorithms are not necessarily sound nor complete and thus do not always return valid results which disqualifies them for direct verification. In this section we want to give a brief insight into a technique that allows to formally verify the correctness independent of the particular algorithm with a verified result checker. Sect. 6.1 outlines the concepts we developed while Sect. 6.2 shows how the result checker can be derived from the specification and be used in the system. For more details, refer to [16].

6.1 Foundations of Verified Result Checking

As a technique to avoid direct verification and as an alternative to pure online verification [25] of the complete system we developed the concept of *verified result checking*, which combines the classical idea of result checking by Blum, Wasserman and Kannan [10, 46] and formal program verification.

Result checking is a way to ensure the correctness of a program by another program. In contrast to testing and verification the correctness is not enforced by ensuring the correctness of the used algorithm itself, but by checking the correctness of all of its results at runtime. To actually be able to give guarantees in advance, we combine the result checking approach with design time verification of the particular result checker. This allows for runtime assurance but design time verification of the algorithm. It also makes it possible to switch algorithms during runtime, e.g., to have specialized algorithms for different situations. The verification task is reduced to the verification of the result checker. The program to check whether a configuration is correct is usually simpler than the program to calculate a configuration. This makes verification of the result checker less complex than the verification of the reconfiguration algorithm.

A result checker is therefore a short program *RC* that reads the output of the program to check and returns `correct` if the result is correct and `incorrect` otherwise (see specification below). It is executed after the reconfiguration algorithm and reads the calculated configuration (V_{conf}). If the configuration restores the invariant, the checker returns `correct` and forwards the configuration. If the configuration is incorrect, it is blocked and feedback is provided to the reconfiguration algorithm. This feedback can consist of the parts of the invariant that are violated or – if a metric is available – a measure of the error of the solution.

Specification of **Result Checker** (*RC*)	
input	configuration (V_{conf}) of o/c_{impl}
output	‘correct’ if $Inv_{RIA}(V_{conf})$, ‘incorrect’, otherwise

While this technique allows to check new configurations for validity, the lack of a direct verification of the algorithm means that no statement can be made about termination of the algorithm. Therefore only correctness and quality properties of a configuration that is forwarded to the system but no liveness criteria in the sense of “something good will eventually happen” about the self-* phase itself can be proven with the result checking approach. Liveness properties ensure that the system makes progress in some manner, while safety properties (“something bad will never happen”) are properties that ensure that there are no threats to life and limb. However, liveness properties in an self-* system are hard to verify as failures can always occur and additional assumptions about the environment have to be made, e.g., assumptions about the frequency of failures. Failures eat redundancy, in the sense of possibilities that another component can take over missing functionality. If no redundancy is available, the system can not compensate a failure any more and therefore, no liveness properties can be guaranteed. They can thus be treated as a kind of quality properties of a considered implementation. This might not be a very relevant limitation in practice but will have to be considered in formal verification. However, the system is still able to guarantee safety properties which are most interesting for self-* systems anyway.

Related Work: A similar idea in order to enable the use of unsound algorithms and still ensure correct results, but not with the focus on formal verification and correctness guarantees, is presented by Rochner and Müller-Schloer in [35]. They add a so called *guard* to their Observer/Controller-architecture that filters the actions calculated by the controller. In principle, this corresponds to the result checking idea. However, they do not consider the correctness of the *guard* itself nor do they describe how such a filter can be derived systematically.

A related field is *runtime verification* [25] which deals with checking the correctness of a property during runtime. This approach tries to completely move the verification from design to runtime, by developing suitable monitors in order to accurately decide whether a property holds or not. In contrast to the approach presented here *runtime verification* does not deal with the question how a system can be adapted if a violation is detected. However, ideas and results from the field can be used to develop appropriate observers in order to detect system failures and invariant violations.

6.2 Deriving und Using a Result Checker

The implementation and the formal specification of a result checker can be systematically derived from the specification of the invariant. The result checker implementation is then verified to prove that it returns `correct` for a configuration if and only if the invariant $INV_{RIA}(conf)$ evaluates to true. In [16] a systematic development and verification of a result checker for self-organizing resource-flow systems is described in detail.

The invariant can be seen as constraints on the configuration variables, as the corridor is formulated as a predicate $INV_{RIA}(V)$ over all variables V. It usually constrains the configuration variables in relation to the remaining variables and therefore describes correct configurations with respect to the system's situation. If the invariant is violated, e.g., due to a failure which is reflected in a change of a variable's value, the observer/controller tries to find a new evaluation for the configuration variables, which re-establishes the invariant.

The reconfiguration task can therefore be considered a constraint satisfaction problem (CSP) [15,44]:

$$CSP_{reconf} := (V_{conf}, D_{conf}, INV_{RIA}(V))$$

The decision variables are the configuration variables V_{conf} with corresponding domains D_{conf} and the constraints are defined through the invariant. The goal is to find a valid evaluation of the configuration variables such that the constraints (invariant) are fulfilled [29]. Usually one is not only interested in whether a solution is correct or incorrect, but also in how good a solution is. In case of an incorrect solution detailed feedback is required in order to find a new and valid configuration. Therefore the result checker can be extended with a penalty function which quantifies the quality of a configuration.

In case of an invalid result the result checker provides feedback about which constraints are violated for which agents. This detailed feedback can than be used by the self-* algorithm to find a better solution. For instance, if a genetic algorithm is used, the result checker can be called by the fitness function and used as one element in the calculation of the fitness values of the configurations in a population.

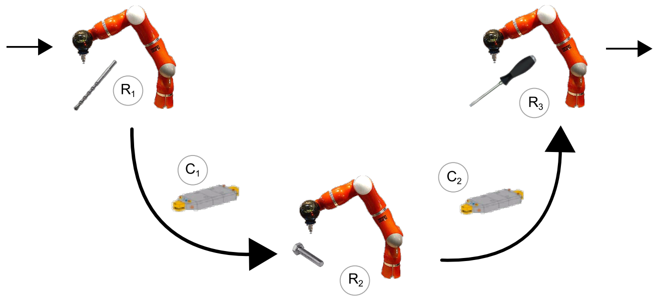


Fig. 10. An adaptive production cell with three robots, two transport units, and three tools [19]

7 Application to an Adaptive Production Cell

In this section, the application of the specification and verification approach to a simple adaptive production cell is described. The cell depicted in Fig. 10 consists of three robots and two autonomous transport units (carts) connecting them. Every robot can accomplish three tasks: drill a hole into a workpiece (D); insert a screw into this hole (I); and tighten the screw with a screwdriver (T). For each task the robots have different tools which they can switch.

Every workpiece entering the cell has to be processed according to a given order, e.g., drill, insert, tighten. In case one or more tools break and the current configuration allows no more correct processing of the incoming workpieces, the observer/controller is reconfiguring the cell and re-assigning the different tools such that production can continue. Further the carts have to be re-routed in order to preserve the right processing sequence. They always have to transport from the drilling robot to the inserting robot and from the inserting robot to the robot that is tightening the screw. As the system is deciding on its own which robot is applying which tool, we at least want to have the guarantee that workpieces are processed correctly: the tools are applied in the right order and workpieces leaving the cell are fully processed with all three tools.

1. Define Formal Model, System Dynamics, and Property. The first step is to build a formal model of the production cell. Each robot has, besides the two flags *reconf* and *deficient* for reconfiguration, a variable *availableTools* describing the set of available tools and a variable *assignedTool* for the currently assigned tool. Access to a variable of a Robot *r* is denoted by *r.assignedTool*. The set *robots* contains all currently participating robots. The task is specified as a sequence of tools which should be applied to a workpiece. A workpiece *wp* therefore has two variables: *wp.state* modeling its current processing state and *wp.task* for the task for this workpiece. The set of workpieces currently in the production cell is denoted with *cell*. In order to formulate properties about workpieces leaving the cell, the leaving workpieces are stored in a *storage* (list of workpieces). The expected property (*Prop*) the system should exhibit then is, that all workpieces have correct state and leaving workpieces are fully processed. It can be formulated as follows, where $\sqsubseteq$ is the standard prefix operator for lists:

$$\begin{aligned} \mathit{correctProcessing}(V_{all}) :\leftrightarrow \quad & (\forall\, wp \in \mathit{cell} : wp.\mathit{state} \sqsubseteq wp.\mathit{task}) \\ & \wedge (\forall\, wp \in \mathit{storage} : wp.\mathit{state} = wp.\mathit{task}) \end{aligned}$$

This property should always be maintained by the complete system and its validity on the entire trace is the behavioral guarantee we want to give for the system.

In [39] an extended variant of the adaptive production cell is described. It includes correct routing of the carts which was not considered here. This leads to additional properties in INV_{RIA}, describing correct routes. Carts are also specified with rely/guarantees. The specification of the corridor and its verification is presented in [32]. Further, the paper contains a role concept for the agents and a detailed model of the dynamics of the agents modeled with UML-statecharts. The implementation of the production scenario is described in [30].

2. Define the Corridor and Reconfiguration Behavior. The invariant specifies correct configurations of the production cell which leads to correct behavior. A valid configuration is one that assigns all needed tools. In other words, for each workpiece in the cell, all tools of the workpiece's task have to be assigned. Further, only tools available to a robot can be assigned.

$$\begin{aligned} INV_{RIA} := \quad & (\forall\, wp \in \mathit{cell}\ \forall\, t \in wp.\mathit{task} : \exists\, r \in \mathit{robots} : r.\mathit{assignedTool} = t) \\ & \wedge (\forall\, r \in \mathit{robots} : r.\mathit{assignedTool} \in r.\mathit{availableTools}) \end{aligned}$$

In this application quiescence means that a robot stops during reconfiguration and does not perform any processing steps, like applying a tool. The observer/controller furthermore guarantees that it does only interfere when a self-* phase was started in beforehand. This means that the o/c does only change a robot's *assignedTool* in functional phases.

3. Instantiate the Abstract Rely/Guarantees. In order to retrieve the rely/guarantee properties the variables have to be assigned to the generic variable sets.

$$\begin{aligned} V_{int} &:= \{r.\mathit{ID} \mid \forall\, r \in \mathit{robots}\} \\ V_{conf} &:= \{r.\mathit{assignedTool} \mid \forall\, r \in \mathit{robots}\} \\ V_{rest} &:= \{r.\mathit{availableTools} \mid \forall\, r \in \mathit{robots}\} \\ V_{env} &:= \{wp.\mathit{state}, wp.\mathit{task} \mid \forall\, wp \in \mathit{cell} \cup \mathit{storage}\} \end{aligned}$$

For this scenario, the robots' internal variables are merely their IDs. The set of configuration variables contains the *assignedTool* variable of each robot. The *availableTools* of each robot can be arbitrarily changed by the environment and are therefore in V_{rest}. The environment is thus allowed to change this set which models tool failures but also maintenance in case the set is extended. From the point of view of the production cell, the task according to which a workpiece should be processed, the storage and the state of the workpieces are in V_{env}. For example, the application of a tool changes the environment as the workpiece's state is manipulated. The task of new workpieces entering the cell is set by the environment.

Observer/controller specification. The next step is to instantiate the generic R/G property of the observer/controller. The observer/controller has to guarantee that it correctly restores the invariant at the end of a self-* phase, indicated by changing the robots *reconf* variable, and that it does not interfere in functional phases. It also guarantees that it does not violate the *correctProcessing* property and that it only changes the configuration variables. The instantiated guarantee then looks like this:

$$\begin{aligned} G_{o/c}(V_{all}, V'_{all}) :\leftrightarrow \quad & (\forall\, r \in robots \;:\; r.reconf \wedge \neg\, r'.reconf \rightarrow INV_{RIA}(V'_{all})) \\ & \wedge (\forall\, r \in robots \;:\; \neg\, r'.reconf \rightarrow r'.assignedTool = r.assignedTool) \\ & \wedge (correctProcessing(V_{all}) \rightarrow correctProcessing(V'_{all})) \\ & \wedge Unchg_{sys}(V_{func} \setminus V_{conf}) \end{aligned}$$

The rely $R_{o/c}$ for the o/c-layer is the same as the generic one which assumes that the functional system does not end the self-* phase on its own. Every implementation, distributed or central, that fulfills this R/G-property is a valid o/c-implementation. For the production cell scenario a central and a distributed o/c-layer was implemented [1, 29]. The central one uses a constraint solver in order to calculate new valid assignments. The distributed one is based on coalition formation and tries to find a minimal set of robots that is able to reconfigure the cell. The correctness in both cases is ensured via a verified result checker (see Sect. 6) which ensures that only correct configurations are forwarded to the functional system.

Functional system specification. The rely/guarantee property for the functional system is retrieved analogously by instantiating the generic property:

$$\begin{aligned} G_{SYS_{func}}(V_{all}, V'_{all}) :\leftrightarrow \quad & (\forall\, r \in robots \;:\; r.assignedTool = r'.assignedTool) && (1) \\ & \wedge (correctProcessing(V_{all}) \rightarrow correctProcessing(V'_{all})) && (2) \\ & \wedge (\forall\, r \in robots \;:\; r.reconf \rightarrow V'_{all} = V_{all}) && (3) \\ & \wedge (\forall\, r \in robots \;:\; r.reconf \rightarrow r'.reconf) && (4) \end{aligned}$$

The functional part does not change the configuration on its own (1) and guarantees the expected property *correctProcessing* (2). It also guarantees to enter the quiescent state, when a self-* phase was initiated (3). The functional system ensures that it only leaves the quiescent state when notified by the o/c (4). To be able to guarantee this, the functional system relies on the o/c not to change the configuration in functional phases. Further it relies on others to not change its internal variables. In this case, the rely is identical to the generic $R_{SYS_{func}}$ shown on page 96.

As the functional system consists of several robots the next step is to split the R/G property into properties for the single robot in a second decomposition step. In this decomposition the interaction between the robots must also be considered, as from the point of view of each individual robot, the o/c as well as the other robots are in its environment. The local rely/guarantees are retrieved by restricting the property to the scope of a single robot. Additionally, each robot has to guarantee that it does not change the variables of the other robots. The local R/G property for a single robot *r* then is:

$$\begin{aligned} G_r(V_{all}, V'_{all}) :\leftrightarrow\ & (r.assignedTool = r'.assignedTool) \\ & \wedge (correctProcessing(V_{all}) \rightarrow correctProcessing(V'_{all})) \\ & \wedge (r.reconf \rightarrow V'_{all} = V_{all}) \\ & \wedge (r.reconf \rightarrow r'.reconf) \end{aligned}$$

The rely is restricted to the local scope analogously.

$$\begin{aligned} R_r(V'_{all}, V''_{all}) :\leftrightarrow\ & (\neg\, r''.reconf \wedge \neg\, r''.deficient \rightarrow Unchg_{env}(V_{conf})) \\ & \wedge (r'.deficient \rightarrow r''.deficient) \\ & \wedge (correctProcessing(V_{all}) \rightarrow correctProcessing(V'_{all})) \\ & \wedge Unchg_{env}(r.ID) \end{aligned}$$

4. Verification. For the particular implementation it must be proven that it is correct with respect to this R/G-specification. Hence, the number of proofs depends on the number of different agent implementations. In case of a homogeneous system consisting of identical agents – like the robots – only one proof has to be made, while in heterogeneous systems in which different agent types can have different dynamics, the proofs have to be performed for each agent type separately.

In this application the functional and self-* phases are alternating. Hence, in the quiescent state the robots come to a full stop while being reconfigured. On the other end of the spectrum of quiescent behavior are systems in which the self-* layer works in parallel with the functional system permanently. In such a case, the o/c-layer is constantly applying new configurations and the requirements for the quiescent state must be less restrictive. Such a system is presented in the next section.

8 Application to Autonomous Virtual Power Plants

Future energy systems require autonomous, decentralized management to deal with the enormous number of power generators and controllable consumers. Decentralized power generation and the limitations of the power network make it necessary to locally manage the balance between power production and consumption in a decentralized fashion. Autonomous Virtual Power Plant s(AVPP) [1, 2] could be the building blocks of such a future system. One AVPP controls a number of small energy producers such as biogas plants, solar plants and run-of-the-river power plants. The plants are divided into stochastic ones such as solar and wind generators and controllable ones. The AVPP's task is to control the plants in such a way that the load equals the combined production of the plants by calculating schedules for the controllable plants. The control decisions are based on forecasts of the plants' power production and of the load. A correct schedule for the individual plants can be calculated by a genetic algorithm or a particle swarm optimizer. The schedule changes as new prognoses come in and old ones are revised. The AVPP thus self-adapts constantly to new information and to the new environmental situation, meaning that there are no strictly discernible self-* and functional phases. In addition, a reactive algorithm compensates for slight errors in the predictions by adapting each power plant's output slightly if current data about production and load become available [3]. However, the calculation of new power plant schedules can be seen as the main self-adaptive feature and is thus designated as the self-* phase in the following.

1. Define Formal Model, System Dynamics, and Property. An AVPP constitutes an observer/controller that manages a functional system consisting of a set N of individual power generators. If a power plant does not produce the power it forecast or the load changes unexpectedly, the invariant of the AVPP is violated and a new schedule has to be calculated. The total load which should be met is denoted as L_c and is available to the observer/controller and all power plants. The AVPP calculates schedules for the power plants based on a load prognosis L_{prog} which approximates the future load. Each power plant $i \in N$ has a scheduled target output $P_{target,i}$ that is derived from the AVPP's target output which in turn is determined by the prognosed load ($P_{target} = L_{prog} = \sum_i^N P_{target,i}$). As the schedule is made for several timesteps in advance, P_{target}, L_{prog} and $P_{target,i}$ are lists of values. The scheduled target output for time t is denoted by $P^t_{target,i}$.

The property (*Prop*) that is of importance in the energy system is grid stability. The power grid is sensitive to imbalances between consumption and production. If they differ, the network frequency changes which can lead to power outages and destroy equipment. Therefore, the AVPP has to guarantee that – if the forecast of the upcoming load is good enough – it will always produce as much power as requested. This boils down to an approximate equality between the scheduled target output of the power plants for the current timestep and their actual output in this time step:

$$gridStability(V_{all}) := P^{now}_{target} \approx \sum_i^N P_{actual,i}$$

Again, it is not sensible to demand strict equality since the prognosis can never be guaranteed to be exactly equal to the actual load. As there is a band in which the power grid can operate and the reactive mechanism can compensate slight deviations, this is not strictly necessary.

2. Define the Corridor and Reconfiguration Behavior. The next step is to formalize the corridor of correct behavior. A valid configuration is one that describes a valid schedule for the system and that ensures that in sum as much power is produced as currently is consumed and that the schedule will be able to cover the consumption predicted for each timestep t.

The first constraint describes that a plant's assigned target output has to be either zero or between the power plant's minimal and maximal output possibilities.

$$C_{cons} : \forall\ i,t : P^t_{target,i} \neq 0 \rightarrow P_{min,i} \leq P^t_{target,i} \leq P_{max,i}$$

Further a valid schedule has to assure that the change of output power from one time step to the next is not greater than the rate of change of a plant (v_i).

$$C_{change} : \forall\ i,t : |P^{t+1}_{target,i} - P^t_{target,i}| \leq v_i$$

In every timestep each plant's target output should be approximately equal to the current output ($t = now$).[3]

$$C_{balanced} := \forall i : P^{now}_{target,i} \approx P_{actual,i}$$

The current power output varies due to the reactive behavior of the power plants. A further constraint describes that the difference between the next target output and the current output may not exceed the rate of change of the plant.

$$C_{variance} := \forall i : |P^{now+1}_{target,i} - P_{actual,i}| \leq v_i$$

The schedule for the stochastic power plants also must assure that the scheduled output approximately equals the forecast:

$$C_{stoch} : \forall\ i,t : P^{t}_{target,i} \approx P^{t}_{pred,i}$$

The corridor is then defined by the conjunction of all the constraints.

$$INV_{RIA}(V) := C_{cons} \wedge C_{change} \wedge C_{balanced} \wedge C_{variance} \wedge C_{stoch}$$

The most common kinds of stochastic power plants are solar and wind power plants. Their output is directly dependent on the weather which thus has to be modeled within the system. It is captured in variables that are combined in the set *Weather*. The output of a stochastic plant is then a function of *Weather*.

The *noInterference* property of the AVPP states that it changes the schedule only after signaling it. The *quiescent state* of a power plant in this case only requires the power plant not to signal the end of a self-* phase itself. This is necessary in order to guarantee that the invariant holds whenever the end of a self-* phase is signaled. It is interesting to note that, in comparison to the agents in the production cell, the power plants can not simply stop whenever a constraint is violated. Instead, in the quiescent state, the power plants stick to their current schedule until a new schedule has been calculated. This behavior, however, has no influence on the verification approach.

3. Instantiate the Abstract Rely/Guarantees. The next step is to assign the variables to the different sets and to instantiate the generic rely/guarantee properties.

$$\begin{aligned} V_{int} &:= \{P_{max,i}, P_{min,i}, v_i, P_{pred,i}\} \\ V_{conf} &:= \{P_{target,i}\} \\ V_{rest} &:= \{P_{actual,i}\} \\ V_{env} &:= \{L_c, Weather\} \end{aligned}$$

The constants describing the physical limitations of the power plants are internal variables, like the maximal possible output. Also the forecast of the future output of a plant is an internal variable. These can not be changed by the environment or the AVPP. The configuration variables consist of the assigned schedule for each power plant. These can not be changed by the environment, but by the AVPP. Each power plant has a variable for the actual power output, which can be changed by the environment. This models failures such as a broken power generator or connection loss to the power grid which lead to a change in the actual output. The environment contains the consumer load and the variables describing the weather.

[3] For the verification this is formalized as the difference may not exceed a certain ε.

Observer/controller specification. Instantiating the generic rely/guarantee properties for the observer/controller we retrieve the specification of the AVPP's self-* behavior.

$$\begin{aligned} G_{o/c}(V_{all}, V'_{all}) :\leftrightarrow \quad & (\forall\, i \,:\, \mathit{reconf}_i \wedge \neg\, \mathit{reconf}'_i \rightarrow \mathit{INV}_{RIA}(V'_{all})) \\ & \wedge (\forall\, i \,:\, \neg\, \mathit{reconf}'_i \rightarrow \mathit{noInterference}(V_{all}, V'_{all})) \\ & \wedge (\mathit{gridStability}(V_{all}) \rightarrow \mathit{gridStability}(V'_{all})) \\ & \wedge \mathit{Unchg}_{sys}(V_{func} \setminus V_{conf}) \end{aligned}$$

The AVPP ensures that it always calculates valid schedules for the system, specified using the invariant $\mathit{INV}_{RIA}(V'_{all})$ and that it does only interfere in self-* phases. The AVPP also guarantees not to violate *gridStability* with its actions. As it does neither produce any power output nor consumes any power and a change of the schedule only comes into effect in the steps of the power plants, this is trivially true. The AVPP only changes the configuration variables, i.e., $P_{target,i}$, of the power plants. It relies on the power plants not to violate *gridStability* either and not to leave the quiescent state during a self-* phase.

R/G-specification of the functional system. The R/G-specification for the functional system is also received by instantiating the generic R/G property. As the functional system is composed of the individual power plants, the second decomposition step is to formulate R/G-Properties for the particular power plant. These local rely/guarantee properties are obtained as in the previous application by restricting the parts which are quantified over all power plants to the particular power plant. In this case, there is no direct interaction between the power plants themselves and thus no additional properties describing such a communication are necessary. The guarantee for a single power plant i then looks like:

$$\begin{aligned} G_i(V_{all}, V'_{all}) :\leftrightarrow \quad & P'_{target,i} = P_{target,i} \\ & \wedge (\mathit{gridStability}(V_{all}) \rightarrow \mathit{gridStability}(V'_{all})) \\ & \wedge (\mathit{reconf}_i \rightarrow \mathit{quiescence}(V_{func}, V'_{func}) \\ & \wedge (\mathit{reconf}_i \rightarrow \mathit{reconf}'_i) \end{aligned}$$

A power plant guarantees not to change the schedule on its own and not to violate the *gridStability* property. The quiescent state of a power plant is that it adheres to the "old" schedule as long as a reconfiguration takes place until the AVPP has finished writing the new schedule and that it does not abort reconfiguration on its own.

In order to be able to guarantee this it must rely on the AVPP not to change the schedule without notification. It further assumes that no internal variables are changed, e.g., the maximal and minimal power output of the plant.

$$\begin{aligned} R_i(V'_{all}, V''_{all}) :\leftrightarrow \quad & (\neg\, \mathit{reconf}''_i \wedge \neg\, \mathit{deficient}''_i \rightarrow \mathit{Unchg}_{env}(P_{target,i})) \\ & \wedge (\mathit{deficient}'_i \rightarrow \mathit{deficient}''_i) \\ & \wedge \mathit{Unchg}_{env}(\{P_{max,i}, P_{min,i}, P_{pred,i}, v_i\}) \end{aligned}$$

4. Verification. For each type of power plant implementation it has to be proven that it satisfies these R/G-properties. Analogously it must be verified that the AVPP's reconfiguration mechanism adheres to its specification and only calculates valid schedules. Then it can be deduced that for a finite but arbitrary number of power plants and an AVPP implementation that adheres to the specification, the *gridStability* property is maintained under the assumption that enough power is available in the whole system to fulfill the load. This rely is formulated in the global environment. In a more detailed model, the AVPP has the ability to throw-off load and therefore to control the load of the system in case the power available is insufficient.

Examples for Invariant Violations. The invariant can be violated in several ways. In case of a stochastic power plant, unforeseen environmental influences such as sudden weather changes can invalidate the forecast output which leads to a violation of C_{stoch} for this power plant. This can lead to a power deficit or surplus, both of which will have to be dealt with by rescheduling the available deterministic plants. A deterministic power plant is less likely to deviate from its prognoses, but it is still possible that the power plant goes offline unexpectedly. In this case, $C_{balanced}$ is violated and other plants have to be rescheduled to compensate for the missing power. A further violation can occur if the load suddenly changes and the reactive mechanism of the deterministic plants changes the actual power output. If this was not foreseen in the prognoses, this leads to a deviation form the scheduled target load of the plant. If this deviation is too big, either $C_{balanced}$ or $C_{variance}$ are violated.

In all cases the AVPP calculates a new schedule that is adapted to the new situation and which fulfills the invariant again. The reactive mechanism of the power plants ensures that grid stability is maintained. An invariant violation shows that the dynamics of the systems are unable to handle the current circumstances and adaptation is needed.

9 Conclusion and Outlook

This chapter presented an approach for formal modeling and compositional verification of self-* systems based on observer/controller-architectures which realize self-* capabilities by adapting configuration parameters of the participating components. The architecture separates the self-* and the functional behavior of the system. This separation is exploited by the Restore Invariant Approach in order to allow a separate verification of the functional and self-* part of the system. For the verification of the functional part a particular behavior of the observer/controller-layer is assumed and vice versa. This is specified by an invariant that defines the corridor of correct behavior in which functional correctness is ensured. The functional system is decomposed into properties over single agents. Compositional reasoning ensures the correctness of the overall system by proofs over single agents.

For the verification of the observer/controller behavior, verified result checking is applied. This allows moving the verification task to design time while correctness is assured during runtime. As correctness is ensured independently of the reconfiguration algorithms, these can be switched during runtime. This also allows the use of algorithms often used in self-* and nature-inspired systems that are neither sound nor complete.

By focusing on a specific architecture, generic properties that all applications have in common can be formulated. These can be instantiated for an application to retrieve the respective proof obligation. For the formal model and its verification, common formalisms and techniques are used that have successfully been applied to traditional systems. This allows to benefit of all advantages, like proof support, but still allows to express the important aspects of self-* systems. The framework includes explicit consideration of the environment, enabling reasoning about feedback loops which are a major aspect when considering self-* systems. However, the explicit model of the environment does not restrict the approach, as unforeseen changes of the environment are implicitly assumed when not specified otherwise. By providing such a fragmentary specification, uncertain behavior or arbitrary behavior can be modeled.

The approach was applied to two applications to illustrate the various aspects and differences, like continuous adaptation without the need of a strong quiescent state compared to alternating phases which need an explicit quiescent state for synchronization and correct reconfiguration.

In conclusion, the presented approach allows giving behavioral guarantees despite the self-* properties of the system. It provides a framework for formal modeling and verification of a system which enables formal proofs without restricting the flexibility of the system to adapt to unforeseen situations. This will hopefully raise the acceptance of self-* systems and facilitates the use of these techniques in safety-critical domains.

Future research will include the consideration of hierarchical architectures with a multi-layered observer/controller structure paired with the functional system, e.g., as presented by Müller-Schloer and Sick in [27]. Such an architecture can be beneficial in the AVPP scenario, where superordinate AVPPs could coordinate the actions of ones located on a lower level. Every level introduces new interaction possibilities that need to be considered in the formal model.

Another interesting extension of the presented approach is to allow the definition of soft corridors where a violation does not cause a reconfiguration right away but allows a fine-grained reaction to problems. The mentioned works in the field of runtime verification could provide useful instructions for the development of monitors in order to recognize a corridor violation quickly. An interesting challenge to solve is to synthesize appropriate monitors based on a given corridor specification and to find methods to decide on a violation early enough.

So far, the verification approach was only applied for safety properties which state that a property is never violated. A next step is to investigate the verification of liveness properties, like e.g., progress properties. This could be done in a similar fashion as presented in [43], where the rely/guarantee approach was already successfully applied to prove liveness properties of lock free algorithms. However, the verification of self-* systems is more complicated, as failures have to be considered. If failures can happen arbitrarily often, there is no possibility to ensure progress of the system. Therefore an approach must be found that restricts the environment, e.g., the frequency of failures. The challenge is to make assumptions that are realistic and still allow failures to happen.

Behavioral guarantees are a major step towards the acceptance of self-* systems and their application in safety-critical domains. Therefore the development of suitable techniques and tools is important. This chapter presented one approach to tackle these challenges. However, it also illustrated how difficult the task is and that there are open questions that will need to be tackled in order to extend the scope of formal techniques to the full range of self-* systems.

Acknowledgments. This work has been sponsored by the German Research Foundation (DFG) in the projects SAVE ORCA as part of the priority program SPP 1183 Organic Computing and ForSA as part of the research unit FOR 1085 OC-Trust. The authors would like to thank Alexander Knapp and Bogdan Tofan for their feedback and the valuable discussions.

References

1. Anders, G., Seebach, H., Nafz, F., Steghofer, J.-P., Reif, W.: Decentralized reconfiguration for self-organizing resource-flow systems based on local knowledge. In: 2011 8th IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems (EASe), pp. 20–31 (April 2011)
2. Anders, G., Siefert, F., Steghöfer, J.P., Seebach, H., Nafz, F., Reif, W.: Structuring and Controlling Distributed Power Sources by Autonomous Virtual Power Plants. In: Proc. of the Power & Energy Student Summit 2010 (PESS 2010), pp. 40–42 (October 2010)
3. Anders, G., Hinrichs, C., Siefert, F., Behrmann, P., Reif, W., Sonnenschein, M.: On the influence of inter-agent variation on multi-agent algorithms solving a dynamic task allocation problem under uncertainty. In: Proceedings of the 2012 Sixth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), IEEE Computer Society, Los Alamitos (2012)
4. Balser, M.: Verifying Concurrent System with Symbolic Execution – Temporal Reasoning is Symbolic Execution with a Little Induction. Ph.D. thesis, University of Augsburg, Augsburg, Germany (2005)
5. Balser, M., Reif, W., Schellhorn, G., Stenzel, K.: KIV 3.0 for Provably Correct Systems. In: Hutter, D., Traverso, P. (eds.) FM-Trends 1998. LNCS, vol. 1641, pp. 330–337. Springer, Heidelberg (1999)
6. Bauer, A., Leucker, M., Schallhart, C.: Runtime Verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. 20(4), 14 (2011)
7. Bäumler, S., Schellhorn, G., Tofan, B., Reif, W.: Proving linearizability with temporal logic. In: Formal Aspects of Computing, FAC (2009)
8. Bäumler, S., Balser, M., Nafz, F., Reif, W., Schellhorn, G.: Interactive verification of concurrent systems using symbolic execution. European Journal on Artificial Interlligence (AI Communication) 23(2-3), 285–307 (2010)
9. Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In: Proc. of the 28th International Conference on Software Engineering (ICSE), Shanghai, China. ACM Press (2006)
10. Blum, M., Kanna, S.: Designing programs that check their work. In: STOC 1989: Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing, pp. 86–97. ACM, New York (1989)
11. Branke, J., Mnif, M., Müller-Schloer, C., Prothmann, H.: Organic Computing - Addressing Complexity by Controlled Self-organization. In: Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2006, pp. 185–191. IEEE (2008)
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.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009)

13. Chandy, M., Misra, J.: An example of stepwise refinement of distributed programs: quiescence detection. ACM Trans. Program. Lang. Syst. 8, 326–343 (1986)
14. De Wolf, T., Holvoet, T.: Designing self-organising emergent systems based on information flows and feedback-loops. In: First International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2007, pp. 295–298 (July 2007)
15. Dechter, R.: Constraint processing. Elsevier Morgan Kaufmann (2003)
16. Fischer, P., Nafz, F., Seebach, H., Reif, W.: Ensuring correct self-reconfiguration in safety-critical applications by verified result checking. In: Proceedings of the 2011 Workshop on Organic Computing, OC 2011, pp. 3–12. ACM, New York (2011)
17. Gärtner, F.C.: Fundamentals of fault-tolerant distributed computing in asynchronous environments. ACM Comput. Surv. 31, 1–26 (1999)
18. Giese, H.: Modeling and Verification of Cooperative Self-adaptive Mechatronic Systems. In: Kordon, F., Sztipanovits, J. (eds.) Monterey Workshop 2005. LNCS, vol. 4322, pp. 258–280. Springer, Heidelberg (2007)
19. Güdemann, M., Ortmeier, F., Reif, W.: Safety and Dependability Analysis of Self-Adaptive Systems. In: Proceedings of ISoLA 2006. IEEE CS Press (2006)
20. IBM: An architectural blueprint for autonomic computing. Tech. rep., IBM Corporation (2006)
21. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. 5(4), 596–619 (1983)
22. Kramer, J., Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management. IEEE Trans. Softw. Eng. 16, 1293–1306 (1990)
23. Kramer, J., Magee, J.: Analysing dynamic change in distributed software architectures. IEE Proceedings Software 145(5), 146–154 (1998)
24. Kramer, J., Magee, J.: Analysing dynamic change in software architectures: A case study, pp. 91–100 (1998)
25. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. 78(5), 293–303 (2009)
26. Misra, J., Chandy, K.M.: Proofs of Networks of Processes. IEEE Transactions on Software Engineering SE-7(4), 417–426 (1981)
27. Müller-Schloer, C., Sick, B.: Controlled emergence and self-organization. In: Organic Computing. Understanding Complex Systems, vol. 21, pp. 81–103. Springer, Heidelberg (2008)
28. Murch, R.: Autonomic Computing. IBM Press (2004)
29. Nafz, F., Ortmeier, F., Seebach, H., Steghöfer, J.-P., Reif, W.: A Universal Self-Organization Mechanism for Role-Based Organic Computing Systems. In: González Nieto, J., Reif, W., Wang, G., Indulska, J. (eds.) ATC 2009. LNCS, vol. 5586, pp. 17–31. Springer, Heidelberg (2009)
30. Nafz, F., Ortmeier, F., Seebach, H., Steghöfer, J.P., Reif, W.: A generic software framework for role-based Organic Computing systems. In: SEAMS 2009: ICSE 2009 Workshop Software Engineering for Adaptive and Self-Managing Systems (2009)
31. Nafz, F., Seebach, H., Steghöfer, J.P., Anders, G., Reif, W.: Constraining Self-organisation Through Corridors of Correct Behaviour: The Restore Invariant Approach. In: Müller-Schloer, C., Schmeck, H., Ungerer, T. (eds.) Organic Computing - A Paradigm Shift for Complex Systems. Autonomic Systems, vol. 1, pp. 79–93. Springer, Basel (2011)
32. Nafz, F., Seebach, H., Steghöfer, J.-P., Bäumler, S., Reif, W.: A Formal Framework for Compositional Verification of Organic Computing Systems. In: Xie, B., Branke, J., Sadjadi, S.M., Zhang, D., Zhou, X. (eds.) ATC 2010. LNCS, vol. 6407, pp. 17–31. Springer, Heidelberg (2010)
33. Pissias, P., Coulson, G.: Framework for quiescence management in support of reconfigurable multi-threaded component-based systems. Iet Software/IEE Proceedings - Software 2, 348–361 (2008)

34. Richter, U., Mnif, M., Branke, J., Müller-Schloer, C., Schmeck, H.: Towards a generic observer/controller architecture for Organic Computing. In: INFORMATIK 2006 – Informatik für Menschen!, vol. P-93, pp. 112–119 (2006)
35. Rochner, F., Müller-Schloer, C.: Emergence in Technical Systems. it - Information Technology 47(4), 195–200 (2005)
36. Schellhorn, G., Tofan, B., Ernst, G., Reif, W.: Interleaved programs and rely-guarantee reasoning with ITL. In: Proc. of Temporal Representation and Reasoning (TIME). IEEE, CPS (2011)
37. Schmeck, H., Müller-Schloer, C., Çakar, E., Mnif, M., Richter, U.: Adaptivity and self-organization in organic computing systems. ACM Trans. Auton. Adapt. Syst. 5, 10:1–10:32 (September 2010)
38. Seebach, H., Nafz, F., Steghöfer, J.P., Reif, W.: A software engineering guideline for self-organizing resource-flow systems. In: IEEE International Conference on Self-Adaptive and Self-Organizing System (SASO), pp. 194–203. IEEE Computer Society, Los Alamitos (2010)
39. Seebach, H., Nafz, F., Steghöfer, J.P., Reif, W.: How to Design and Implement Self-organising Resource-Flow Systems. In: Müller-Schloer, C., Schmeck, H., Ungerer, T. (eds.) Organic Computing - A Paradigm Shift for Complex Systems, Autonomic Systems, vol. 1, pp. 145–161. Springer, Basel (2011)
40. Shehory, O., Kraus, S.: Methods for task allocation via agent coalition formation. Artificial Intelligence 101(1-2), 165–200 (1998)
41. Smith, G., Sanders, J.W.: Formal Development of Self-organising Systems. In: González Nieto, J., Reif, W., Wang, G., Indulska, J. (eds.) ATC 2009. LNCS, vol. 5586, pp. 90–104. Springer, Heidelberg (2009)
42. Sterman, J.D.: Business Dynamics – Systems Thinking and Modeling for a Complex World. McGraw-Hill (2000)
43. Tofan, B., Bäumler, S., Schellhorn, G., Reif, W.: Temporal Logic Verification of Lock-Freedom. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 377–396. Springer, Heidelberg (2010)
44. Tsang, E.: Foundations of Constraint Satisfaction. Computation in Cognitive Science. Academic Press, Inc., London and San Diego, USA (1993)
45. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: An alternative to quiescence: Tranquility. In: 22nd IEEE International Conference on Software Maintenance, ICSM 2006, pp. 73–82 (September 2006)
46. Wasserman, H., Blum, M.: Software reliability via run-time result-checking. J. ACM 44(6), 826–849 (1997)
47. Wooldridge, M.J., Dunne, P.E.: The Computational Complexity of Agent Verification. In: Meyer, J.-J.C., Tambe, M. (eds.) ATAL 2001. LNCS (LNAI), vol. 2333, pp. 115–127. Springer, Heidelberg (2002)
48. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: Proceedings of the 28th International Conference on Software Engineering, ICSE 2006, pp. 371–380. ACM, New York (2006)
49. Zhang, J., Goldsby, H.J., Cheng, B.H.: Modular verification of dynamically adaptive systems. In: Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development, AOSD 2009, pp. 161–172. ACM, New York (2009)

Timed Hazard Analysis of Self-healing Systems

Claudia Priesterjahn[1], Dominik Steenken[2], and Matthias Tichy[3]

[1] Heinz Nixdorf Institute, Software Engineering Group, University of Paderborn
[2] Research Group Specification and Modelling of Software Systems, Department of Computer Science, University of Paderborn
[3] Software Engineering Division, Chalmers University of Technology and University of Gothenburg, Sweden
{cpr,dominik}@uni-paderborn.de, tichy@chalmers.se

Abstract. Self-healing can be used to reduce hazards in embedded real-time systems which are applied in safety-critical environments. These systems may react to failures by a structural reconfiguration of the architecture during runtime. This means the exchange of components or the modification of the components' connections, in order to avoid that a failure results in a hazard. This reaction is subject to hard real-time constraints because reacting too late does not yield the intended effects. Consequently, it is necessary to analyze the propagation of failures in the architectural configuration over time with respect to the structural reconfiguration. However, current approaches do not take into account the timing properties of the failure propagation and the structural reconfiguration. In this paper, we present a hazard analysis approach which specifically considers these timing properties. We illustrate our approach by an example case study from the RailCab project. Further, we demonstrate the scalability of the approach by experiments.

Keywords: Hazard Analysis, Real-time, Reconfiguration, System Safety.

1 Introduction

Safety-critical systems like cars, airplanes, trains, and medical devices are subject to rigorous analyses to ensure safety of persons, goods or the environment [23,40]. Hazard analysis approaches play a key role as hazards which might lead to accidents have to be identified and handled accordingly.

Hazard analysis approaches typically follow either a bottom-up approach, e.g., Failure Mode Effect Analysis (FMEA), or a top-down approach, e.g., Fault Tree Analysis (FTA). In the last couple of years, the trend has been to exploit the component structure of safety-critical systems to improve the hazard analysis [12, 15, 16, 18, 22, 30, 44].

These hazard analysis approaches require that the component structure does not change at runtime as they do not account for different architectural configurations (from now on configuration). However, this prevents their application to systems which employ self-* techniques which has become a major trend in engineering complex systems (cf. [7]) as these systems change their behavior, e.g., by structural reconfiguration of the configuration [27], at runtime.

Self-* systems adapt autonomously to changes in the system itself, such as failing physical components, or in the environment, such as unexpected external stresses on the system. For safety-critical systems, self-healing is one specific self-* technique that can be used to reduce the probability of hazards. This is achieved by an appropriate reaction after detecting an error to avoid the hazardous situation.

We propose to use particularly the structural reconfiguration of the configuration as a way to stop the propagation of failures through the component structure [3] before they result in a hazard. This is achieved by reconfiguring intentionally into an appropriate configuration. This can be achieved, e.g., by disconnecting a failed subconfiguration and starting other components. Our structural reconfiguration allows for modifying complex configurations in a flexible manner. This is achieved by specifying structural reconfigurations by graph transformations [11].

In order to judge whether a reconfiguration reduces the probability of the hazard successfully, we must consider that failures continue to propagate even during the reconfiguration [31]. Therefore, a correct hazard analysis of self-healing systems must take the propagation times of failures, the duration of the reconfiguration, and the change that results from the reconfiguration into account.

Current automated hazard analysis approaches [1,8,14,17,24,43] take either the propagation times of failures or the effects of structural reconfiguration on the failure propagation into account, but not both. They are thus not suitable for the hazard analysis of self-healing systems.

In this paper, we present a timed hazard analysis for reconfigurable systems that employ self-healing (extending a previous approach in [33]). This approach particularly considers timing characteristics of the failure propagation as well as the reconfiguration. We employ formal specifications for the timed structural reconfiguration based on the graph transformation formalism and for the timed failure propagation in the architecture. Furthermore, we use timed automata for the specification of the behavior of components.

Our timed hazard analysis checks whether the structural reconfigurations are executed fast enough such that a hazard is avoided successfully. We restrict the expensive analysis of the time properties to a small part of the component structure by exploiting the fact that graph transformations are applied locally. By this, we improve the scalability of the timed hazard analysis. The scalability is shown by an experiment.

In the next section, we give an overview of our timed hazard analysis. In Section 3, we present our running example. We then introduce the model used for timed hazard analysis in Section 4. These are the models for the system architecture and the reconfiguration as well as the modeling of timed failure propagation and its formalization. The timed hazard analysis approach in Section 5 then uses these formal models to gauge the effectiveness of any given reconfiguration with respect to hazard reduction. We show results from the simulation experiments in Section 6. Section 7 contains a discussion of related work. We conclude and give an outlook on future work in Section 8.

2 Overview of the Approach

Our timed hazard analysis checks for a given self-healing operation whether this operations fulfills the self-healing successfully or not. This means, it checks whether a failure is stopped on time to prevent the hazard from occurring. We therefore assume that the self-healing action is given before our analysis is applied.

Our timed hazard analysis follows the terminology of Laprie et al. [3]. *Failures* are the externally visible deviation from the component's behavior. They are associated with ports where the component instances interact with their environment. *Errors* are the manifestation of a *fault* in the state of a component, whereas a fault is the cause of an error. Errors are restricted to the internals of the component.

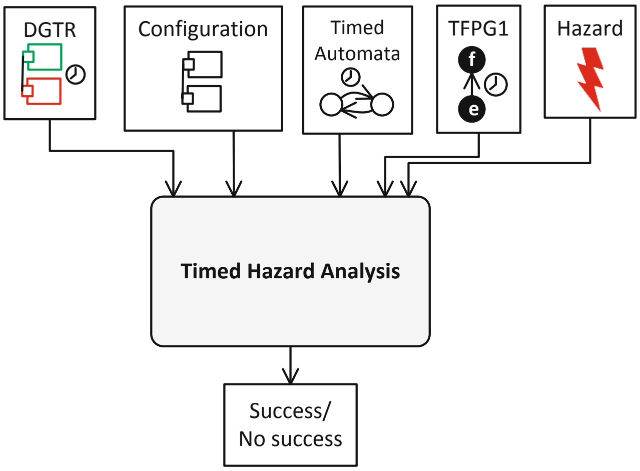


Fig. 1. Overview of timed hazard analysis

Fig. 1 shows an overview of our timed hazard analysis. In the beginning the following models are given: A system in form of a configuration that specifies a component-based system architecture, the behavior of each component defined by a timed automaton, the failure propagation of each component specified by a Timed Failure Propagation Graph[1] (TFPG), a hazard, and the self-healing

[1] A Timed Failure Propagation Graph specifies the propagation of failures through components and the propagation times of failures between different nodes in the system.

action in form of a Durative Graph Transformation Rule[2] (DGTR). All these models need to be specified manually by the developer. TFPGs may be generated automatically from timed automata by our approach published in [32]. However, this is out of scope of this paper.

Timed hazard analysis computes automatically the success of the self-healing operation given by the DGTR on the given system. First, the minimal cut sets (MCS)s[3] of the given hazard are computed. For all these MCSs, it is checked how far the errors of each MCS may propagate through the system after they have been detected and before the reconfiguration is executed. This is done by reachability analysis on Time Petri Nets [36]. Time Petri Nets are the underlying semantics of TFPGs. The result is the state of errors and failures in the system at the time that the reconfiguration is executed. The timed hazard analysis then determines how the reconfiguration effects this state of errors and failures. Finally, it checks whether the errors and failures that remain in the system after the reconfiguration still cause the hazard. If they can not lead to a hazard anymore, the self-healing is successful in reducing the hazard.

3 Running Example

Our running example is adapted from the RailCab[4] – a rail vehicle that is developed by the RailCab project at the University of Paderborn. The vision of the RailCab project is a mechatronic rail system where autonomous vehicles called RailCabs apply the linear drive technology, as used by the Transrapid system, but travel on the existing passive track system of a standard railway system. This system is currently under construction and a first version of the controlling software of the physically existing system has been built using the approach of [19]. RailCabs drive in a convoy in order to reduce energy consumption caused by air resistance and to achieve a higher system throughput. Such convoys require small distances between the RailCabs. These required small distances make the drive in convoy a very safety-critical operation. The correct speed of all vehicles of a convoy must be guaranteed. Further, contrary to regular rail vehicles, the RailCab has an active steering. When driving around curves and over switches, the RailCab has to steer accordingly. The active steering is also used to reduce wheel flange abrasion.

We illustrate our timed hazard analysis by a simplified subsystem of the RailCab. The configuration is depicted in Fig. 2. The subsystem consists of the two subsystems Speed and Steering that control the RailCab's driving speed and steering angle. This subsystem is a safety-critical part of the RailCab. A wrong speed or steering angle can lead to harmful accidents like derailment. The architecture of this subsystem has been adapted from earlier works [19].

[2] A Durative Graph Transformation Rule defines a structural reconfiguration and the time needed to execute this reconfiguration.

[3] Cut sets are combinations of errors that lead to a hazard. MCS are a minimum combination of errors that lead to a hazard. If one error is removed from a MCS, the hazard can not occur.

[4] `http://www.railcab.de`

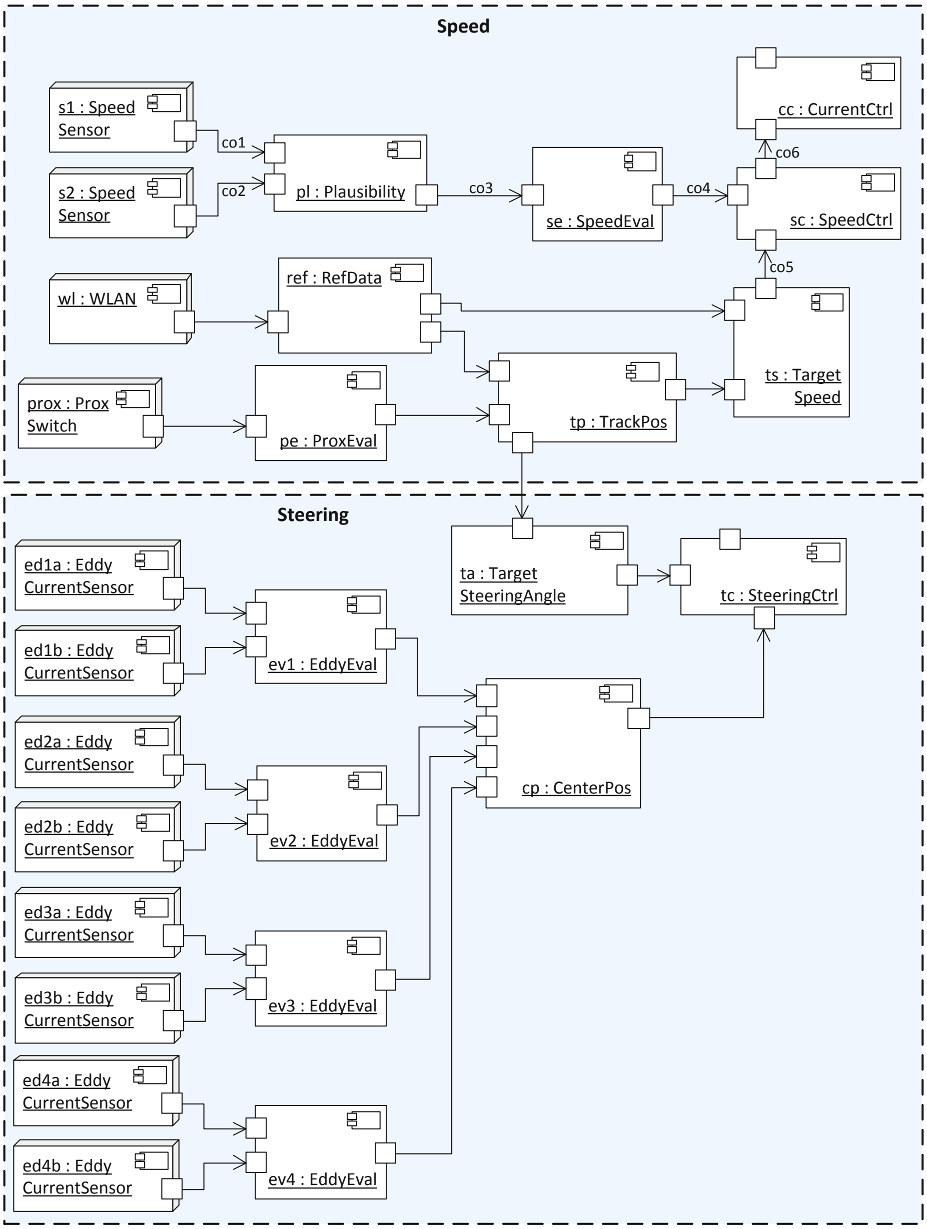


Fig. 2. Start Configuration of the Application Example

The subsystem Speed measures the current speed with two independent speed sensors s1:SpeedSensor and s2:SpeedSensor. These two signals are read by pl:Plausibility, which computes the difference between them. If the difference is too high, the system reconfigures such that the speed sensors are replaced by GPS. Otherwise, the values are forwarded to se:SpeedEval which calculates the current driving speed. The current driving speed is forwarded to sc:SpeedCtrl which calculates the target electric current. This target electric current is forwarded to cc:CurrentCtrl which computes the electric current to be set on the engine in order to reach the target speed.

The subsystem ts:TargetSpeed computes the target speed under the consideration of reference data, such as the target speed and target position provided by the vehicle driving in front, and the current longitudinal position of the RailCab on the track. The reference data is provided by the components wl:WLAN and ref:RefData via wireless network. Further, the position is updated every 15 meters by a proximity switch prox:ProxSwitch which is evaluated by pe:ProxyEval.

The subsystem Steering uses eddy current sensors[5], to measure the lateral position of the RailCab on the track. Each of the four wheels has a pair of eddy current sensors, e.g., ed1a:EddyCurrentSensor and ed1b:EddyCurrentSensor. Each pair is evaluated by a software component, e.g., ev1:EddyEval. The pairs of eddy current sensors are interconnected diagonally such that cp:CenterPos calculates the lateral position of the RailCab by intersecting the two resulting diagonals as shown in Fig. 3.

Based on the input of tp:TrackPos, ta:TargetSteeringAngle calculates a target steering angle. The component sc:SteeringCtrl determines the necessary control values for the steering actuators based on the target steering angle input from ta:TargetSteeringAngle and the RailCab's position on the track input from cp:CenterPos.

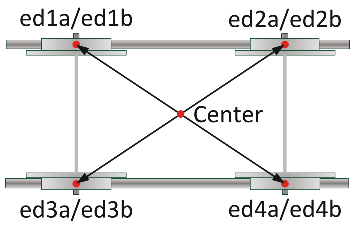


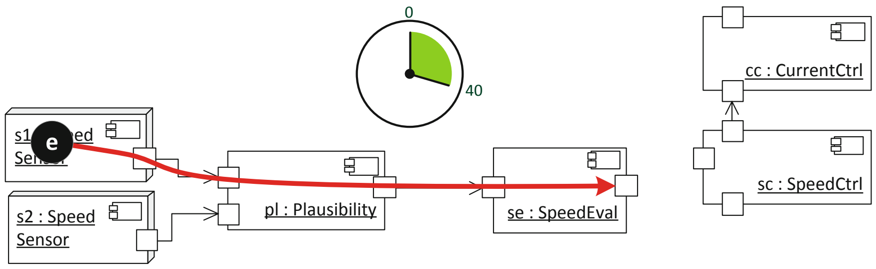
Fig. 3. Schema of Calculation of Center Position

It is a hazardous situation if the RailCab is driving at a wrong speed, e.g., if cc:CurrentCtrl outputs wrong values. A wrong speed could lead to derailment or a collision with another vehicle. This is particularly dangerous, if the RailCab

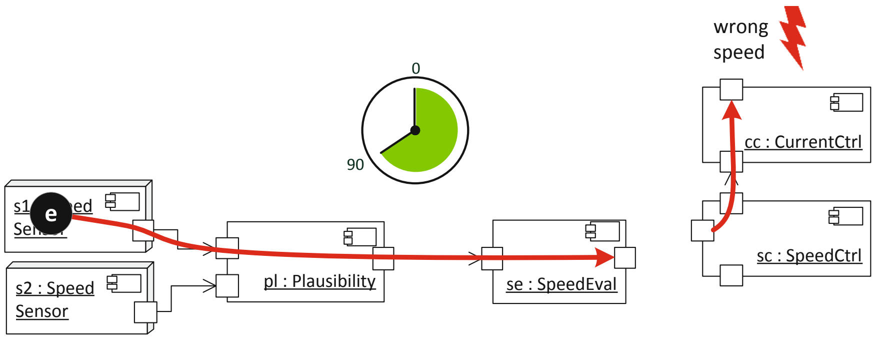
[5] Eddy current sensors measure the field strength of the electric field that is induced by moving an electrically conductive body in a magnetic field.

is part of a convoy which implies low distances of less than one meter between the vehicles. A wrong speed may occur if at least one of the speed sensors, e.g, s1:SpeedSensor emits wrong values.

In order to prevent this hazard, the system detects the error by checking the difference between the values of the two speed sensors. If this difference becomes too high, the system reacts by deleting the connector between se:SpeedEval and ts:TargetSpeed and switching on a GPS subsystem[6] that now measures the RailCab's driving speed.



(a) Reconfiguration at a Clock Value of 40 Time Units



(b) Reconfiguration at a Clock Value of 90 Time Units

Fig. 4. Failure Propagation during Reconfiguration

Figure 4 shows the effects of this reconfiguration for different points of time. In Fig. 4(a) the reconfiguration is executed 40 time units after the detection of the error in s1:SpeedSensor. The failure has not propagated to the components ts:TargetSpeed and sc:SpeedCtrl as the reconfiguration has changed the structure before this happens. In Fig. 4(b), the reconfiguration is executed 90 time units after error detection. Even though the reconfiguration has been executed, the

[6] The GPS subsystem is not active before significant errors in the speed sensors are detected. GPS is less reliable than the speed sensors, because it only works if the satellites are visible. Satellites are not visible if the RailCab enters a tunnel. We consider the GPS-mode as degraded mode.

failure has already propagated to the component sc:SpeedCtrl and caused the hazard.

To prevent the latter case, our hazard analysis takes time properties into account. The basic idea here is that the propagation of a failure can be stopped by reconfiguring the component structure into another component structure in which the error does not cause a hazard. For that to be possible, the reconfiguration has to be faster than the propagation of this error.

4 Modeling the System

In this section we present our models that specify the system architecture, the reconfiguration of the system architecture and the system behavior. With our system model we strive to provide a sound formal foundation on which we will build our hazard analysis. The starting point for this is UML.

From UML component and deployment diagrams we derive our concept of configuration that forms the static part of the system model. For the dynamic part, we consider two types of behavior: reconfiguration and stateful internal behavior / message passing. For reconfigurations we use extended graph transformation rules that we apply to the graphs that constitute component structures. For modeling the internal behavior of the component instances and their communication with each other we use networks of timed automata [2].

4.1 System Architecture

For our system architecture, we focus on the structural view, namely components and connectors. A component encapsulates its inner structure and behavior and allows interaction with other components only via its interfaces. In our component model, the interfaces are realized by ports. We distinguish component types and component instances. Component types define a component with ports that can be used to derive a number of component instances. Each component type can be instantiated multiple times. Note that ports are never instantiated on their own. They are rather instantiated as adjuncts of their respective component type.

A configuration is a concrete assembly of component instances, connected by connectors at the port instances, which we illustrate by UML component diagrams. A configuration of our running example[7] is shown in Fig. 2. Due to reconfiguration a system has several possible configurations.

We will define the necessary elements of our approach formally in order to put our approach on a solid mathematical footing. In choosing and developing these definitions, we will strive to keep as close to our immediate needs as possible to facilitate easy implementation and local proofs. This means that we use a comparatively high number of definitions that are tailored to the specific needs

[7] For later usage, some connectors are named.

of each part of our approach. While there is merit to the idea of creating a more comprehensive formalism that encompasses the entire approach using a small, unified set of definitions, our immediate goals conflict with this idea. We therefore leave it as future work.

We start with the definition of the system architecture, called configurations. Each configuration is based on a component specification. We define a component specification as a type graph as defined in [11].

Definition 4.1 (Component Specification)
The component specification *sys is a type graph [11] sys* $= (V_{sys}, E_{sys}, s_{sys}, t_{sys})$ *over a set of* component types K*, a set of* port types P*, and a set of* connector types L *with*

- $V_{sys} = K \cup P \cup L$,
- $s_{sys} : E \mapsto V$ *the source function, and*
- $t_{sys} : E \mapsto V$ *the target function.*

Thus, a component specification specifies a system in that it determines what component, port and connector types exist, what port types are associated with these component types, and which port types can be connected by which connectors. A particular *configuration*, i.e. a runtime state, of a system is then defined by a set of *instances* of these types, together with a relation showing interconnection between port instances. We define a configuration as a typed graph as defined in [11] based on the type graph of the component specification.

Definition 4.2 (Configuration)
We define $\overline{K}$ *the set of* component instances *of* K*,* $\overline{P}$ *the set of* port instances *of* P*, and* $\overline{L} \subseteq \overline{P} \times \overline{P}$ *the set of connector instances of* L*. Let* $sys = \{V_{sys}, E_{sys}, s_{sys}, t_{sys}\}$ *be a component specification.*

A configuration $w = (G_{conf}, type)$ *of a component specification sys is a typed graph, where* $G_{conf} = (V_{conf}, E_{conf}, s_{conf}, t_{conf})$ *is a graph with the vertices* $V_{conf} = \overline{K} \cup \overline{P}$*, the edges* $E_{conf} = \overline{L}$*, and the source and target function* s_{conf} *and* t_{conf} *and type is a graph morphism typing* G_{conf} *over sys.*

The set of all configurations of a component specification sys is denoted by W_{sys}*. By k.p, we denote port type p of component type k. The same notation holds for component instances.*

4.2 Structural Reconfiguration

For the definition of our structural reconfiguration we use *durative graph transformation rules* (DGTR) [10]. Reconfiguration means changing the set of component instances and their interconnections at runtime. Thus, a reconfiguration transforms one configuration into another configuration. A DGTR is a classic graph transformation rule (cf. [10, 11]) extended by a time interval that specifies the minimum and maximum time needed to execute the DGTR.

Any DGTR consists of two graphs, a *left hand side*(LHS) and a *right hand side*(RHS). In our case, each of the graphs represents a subconfiguration [41].

In order to achieve a succinct notation for a DGTR, we merge both the LHS and the RHS into one graph, annotating the parts occurring in the LHS but not in the RHS with the stereotype «`destroy`» and elements occurring in the RHS but not in the LHS by «`create`». These stereotypes are implicitly carried over to the sub objects of an element, e.g., the deletion of a component results in the deletion of the component's ports, too.

Figure 5 shows the DGTR for our example. The DGTR specifies that the connector between instances of the component types SpeedEval and SpeedCtrl including the corresponding ports are deleted. On the other hand, the component instances GPS:GPS and ge:GPSEval, a connector between them, and a port at ts:TargetSpeed including a connector between ge:GPSEval and ts:TargetSpeed are created. Figure 5 also shows the matching of the ports represented by dashed arrows. This matching is part of the DGTR. Note that for the sake of clarity we have chosen a simple example that is easy to follow.

In addition to the LHS and RHS, each DGTR has a time interval, called *duration*, that specifies the minimum and maximum time units it takes to apply the DGTR to a configuration during runtime. This duration includes the time for created components to start up and for deleted components to shut down.

The time interval in Fig. 5 specifies that at least 30 and at most 42 time units elapse from the time when the application of the rule is initiated and the last change in the configuration that completes the rule. We assume that this information is known by the system developer. We now give a formal definition of a DGTR.

Definition 4.3 (Durative Graph Transformation Rule)
A durative graph transformation rule *(DGTR) is a tuple* $r = (LHS, RHS, d)$*. LHS and RHS are typed graphs as defined in [11]. $d = [d_{min}, d_{max}]$ with $d_{min}, d_{max} \in \mathbb{R}_{\geq 0}$, $d_{min} \leq d_{max}$ is the duration of time needed to transform LHS into RHS.*

In our approach, DGTRs are defined for component types. This means, DGTRs are specific to the component specification of a system but not specific to a configuration. DGTR application, of course, takes place on component instances rather than component types.

We assume that each DGTR has its own clock that starts at zero, when the execution of the DGTR is started. The duration interval of the DGTR refers to this internal clock.

The application of a DGTR to a graph requires to identify a matching of the DGTR's left hand side to the configuration.

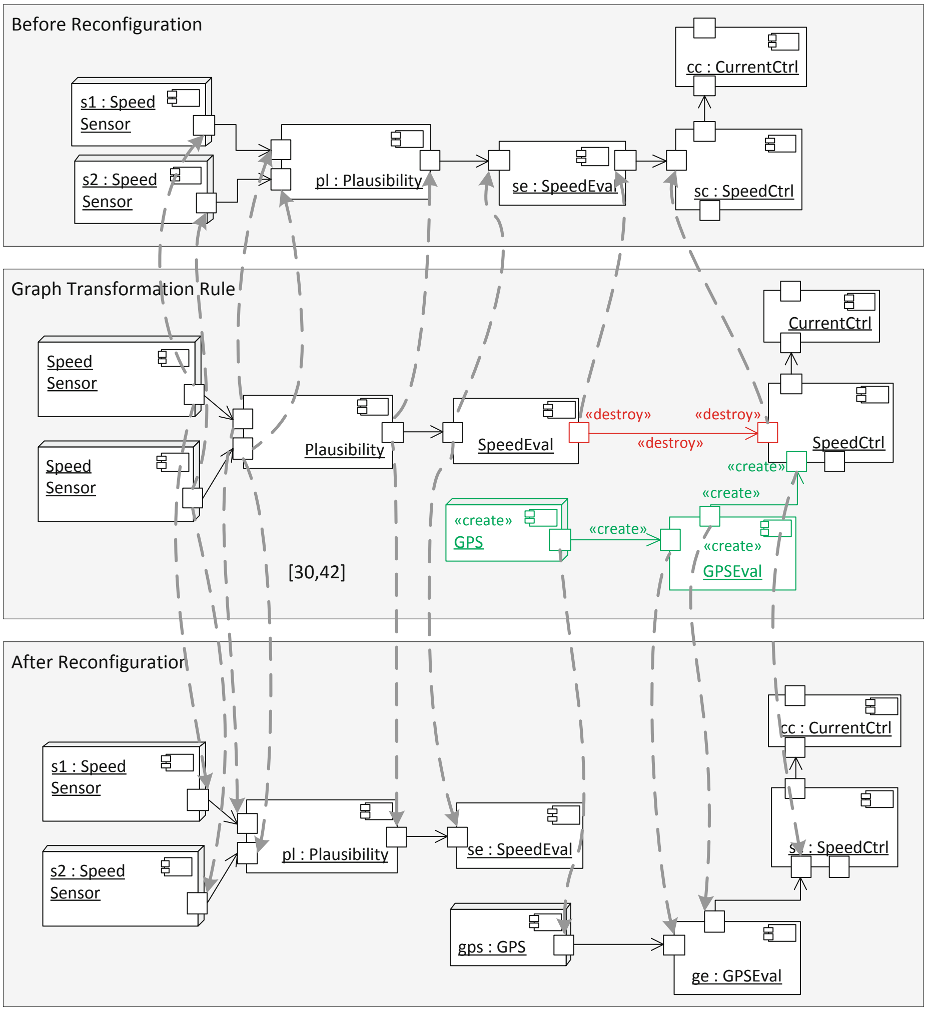


Fig. 5. Application of a DGTR on a configuration

Definition 4.4 (Matching)
Let c be a configuration and $r = (LHS, RHS, d)$ a DGTR both typed over a component specification sys. We define a matching $m(c, r)$ *of the DGTR r on the configuration c as a subgraph isomorphism [11] of LHS on c.*

Matchings require to be isomorphic. When using a homomorphism, it is possible to match two distinct nodes in the left hand side to the same node of the configuration. This may lead to undesired behavior if for example the DGTR defines that one of these nodes should be preserved while the other should be deleted (cf. [11]).

We assume that no concurrent reconfigurations occur, i.e., all aspects of the reconfiguration are described as single DGTRs or sequences of DGTRs. This is to exclude the possibility of unintentional interference between DGTRs that might lead to arbitrary intermediate configurations or an over- or underestimation of the execution time.

The duration of the DGTR only specifies the minimum and maximum time needed to execute all its operations. Consequently, we do not know, which operation of the DGTR is executed at which point in time. Therefore, we assume the worst case for our analysis: The DGTR is executed at the latest point of time possible and all operations are executed in zero time. With this assumption, the failures can propagate the farthest possible through the system before the reconfiguration is applied.

4.3 System Behavior

We model the behavior of software components by timed automata as defined in [4]. A timed automaton is a finite automaton that is extended by a set of real-valued variables called clocks. These clocks measure the elapse of time. By using timed automata, the developer defines time dependent behavior. Thus, the behavior of the component does not only depend on its input, but also on the point in time at which these inputs are received. This is essential for modeling real-time systems.

In order to maintain a separation of concerns [26], i.e., not to mix the component's functional behavior with its self-healing behavior, the behavior of each component is specified by at least two timed automata. One timed automaton specifies the functional behavior. The other specifies the self-healing behavior, i.e., error detection and the resulting initiation of reconfiguration.

For the definition of timed automata, we require a definition of clock constraints first. They are used to model conditions on the values of clocks in a timed automaton.

Definition 4.5 (Clock Constraint)

Let C be a set of real-valued clocks. A clock constraint B is a conjunctive formula of atomic constraints of the form $x \sim n$ or $x - y \sim n$ for $x, y \in C$, $\sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. We use $\mathcal{B}(C)$ to denote the set of clock constraints. [4]

The set of clock constraints is now used in the definition of a timed automaton. Based on the clocks, a timed automaton specifies time guards, clock resets, and invariants. A time guard is a clock constraint that restricts the execution of a transition to a specific time interval. A clock reset sets the value of a clock back to 0 while a transition is fired. Invariants are clock constraints associated with locations that forbid that a timed automaton stays in a location when the clock values exceed the values of the invariant. We extend the timed automata of [4] by side effects that represent the execution of methods for reconfiguration.

Definition 4.6 (Timed Automaton)

A timed automaton A is a tuple $A = (L, l_0, C, \Sigma, R, E, I)$ where

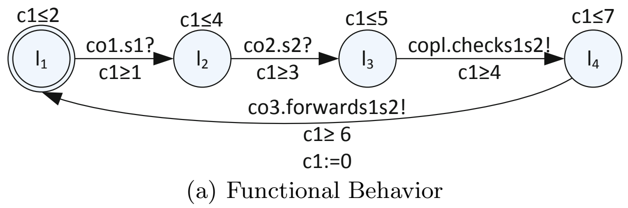
- *L is a finite set of locations,*
- *$l_0 \in L$ is the initial location,*
- *C is a finite set of clocks,*
- *$\Sigma = (D \times \{?, !\}) \cup \{\tau\})$ is a finite set of actions where D is a set of action names and τ is the empty action,*
- *R is the set of side effects, $\epsilon \in R$ is the empty side effect,*
- *$E \subseteq L \times \mathcal{B}(C) \times \Sigma \times R \times 2^C \times L$ is the set of edges where $\varphi \in \mathcal{B}(C)$ is the transition guard and $\lambda \in 2^C$ are the clock resets, and*
- *$I : L \rightarrow \mathcal{B}(C)$ assigns clock constraints to locations, the invariants.*

We shall write $l \xrightarrow{\varphi,a,r,\lambda} l'$ when $(l, \varphi, a, r, \lambda, l') \in E$.

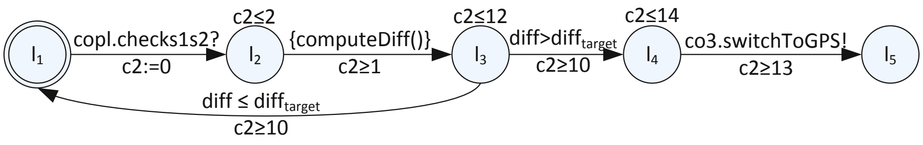
In addition to time guards and resets, a transition may carry an action symbol of Σ that specifies input actions and output actions of the timed automaton. Input actions are denoted by ?, output actions by !. In order that only actions are transmitted between the TA of components that are connected by a connector, we relate actions to connectors. We thus use actions of the form *co.m*. *co* specifies the name of the connector and *m* the message name.

Figure 6 shows an example of timed automata. The TA of Fig. 6(a) specifies the functional behavior of the component type Plausibility. The TA of Fig. 6(b) specifies the self-healing behavior of the component type Plausibility.

The timed automaton of Fig. 6(a) consists of four locations l_1 to l_4 and four transitions connecting the locations. l_1 is the starting location. Starting locations are marked by a by double circles. The invariant $c1 \leq 2$ of location l_1 specifies



(a) Functional Behavior



(b) Self-healing Behavior

Fig. 6. Timed Automata of the Component Type Plausibility

that l_1 may only be active while the value of $c1$ is less than or equal to 2. Accordingly, the time guard $c1 \geq 1$ restricts the firing of the transition to a minimal value of 1 of clock l_1. The term $c1 := 0$ at the transition from l_4 to l_1 sets the clock $c1$ back to 0.

The timed automaton of Fig. 6(b) receives the sensors values $s1$ and $s2$ by the action $copl.checks1s2?$. Then, it computes the difference between these values. If the difference is lesser than or equal to acceptable difference $diff_{target}$, the timed automaton switches back to the initial state. Otherwise, a failure has been detected. In this case the self-healing is triggered by the action $co3.switchToGPS!$.

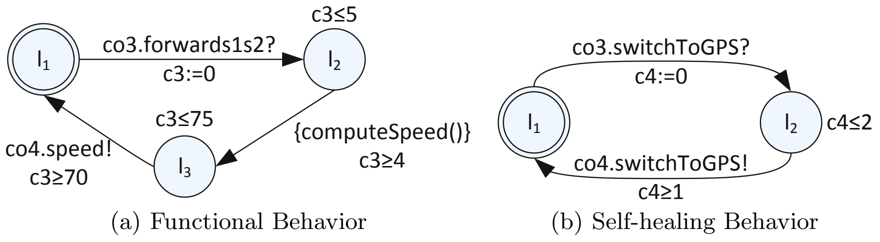


Fig. 7. Timed Automata of the Component Type SpeedEval

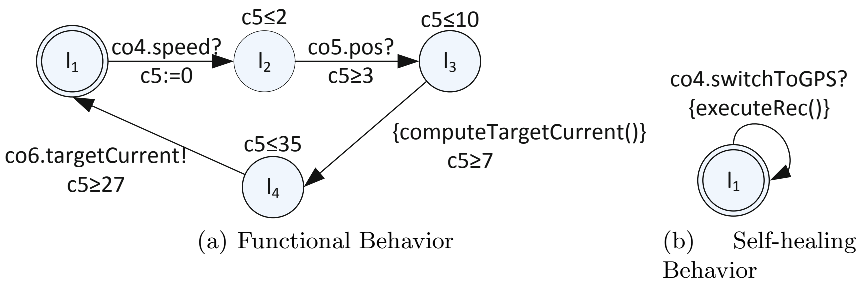
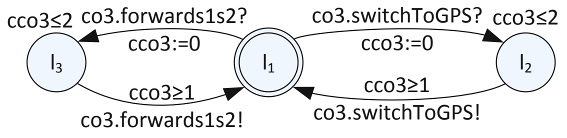


Fig. 8. Timed Automata of Component Type SpeedControl

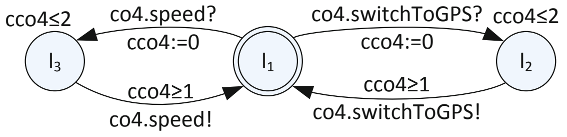
Figure 7 shows the TA that specify the functional behavior (cf. Fig. 7(a)) and the self-healing behavior (cf. Fig. 7(b)) of the component type SpeedEval. Figure 8 shows the TA that specify the functional behavior (cf. Fig. 8(a)) and the self-healing behavior (cf. Fig. 8(b)) of the component type SpeedCtrl.

For each connector, we create a timed automaton that models the delay that is introduced by the connector. For component-internal communication each component has a component-internal connector that has no delay. It thus is not modeled as a timed automaton.

Figure 9 shows the timed automata that model the behavior of the connectors between pl:Plausibility and se:SpeedEval (cf. Fig. 9(a)), and between se:SpeedEval



(a) Behavior of Connector co3



(b) Behavior of Connector co4

Fig. 9. Timed Automata Specifying Connector Behavior

and sc:SpeedCtrl (cf Fig. 9(b)). The transition from l_1 to l_2 in the timed automaton of Fig. 9(a) receives the action $co3.switchToGPS?$. The timed automaton then waits for a maximum of 2 time units in l_2. The transition from l_2 to l_1 is fired the earliest when clock $cco3$ has a value of 1. Then, action $co3.switchToGPS!$ is sent. Thus, the timed automaton of Fig. 9(a) models a minimum delay of 1 time unit and a maximum delay of 2 time units.

In our example, the input action $co1.s1?$ specifies that message $s1$ is received via connector $co1$. $co1$ is the connector that connects s1:SpeedSensor and pl:Plausibility. The output action $co3.forwards1s2!$ specifies that $forwards1s2$ is sent via connector $co3$. The action $copl.checks1s2!$ of the TA of Fig. 6(a) is sent via the component-internal connector $copl$ to the TA of Fig. 6(b) in the same component. Note that we abstract from real sensor data in our behavioral models.

Given the timed automata of Fig. 6, they can now interact with each other using the joint set of actions Σ. Such a set of interacting timed automata is referred to as a *network of timed automata* [4]. We formalize networks of timed automata as follows.

Definition 4.7 (Network of Timed Automata)
A network of timed automata NTA consists of $n \in \mathbb{N}$ timed automata $A_1, \ldots, A_n$ with $n \geq 2$. For all $A_i, A_j \in NTA$ with $i, j \in \{1, \ldots, n\}, i \neq j$: $L_i \cap L_j = \emptyset$, $C_i \cap C_j = \emptyset$ and $\Sigma_i = \Sigma_j = \Sigma$.

The TA of Fig. 6-8 build an NTA as they communicate via synchronizations. The TA of Fig. 6(a) and 6(b) for example synchronize via copl.checks1s2. With this the TA of Fig. 6(a) forwards the data of the speed sensors s1:SpeedSensor and s2:SpeedSensor to the TA of Fig. 6(b). Before, the TA of Fig. 6(a) has received the values from the speed sensors by co1.s1? and co2.s2?. After this, the TA of Fig. 6(a) forwards the values $s1$ and $s2$ to SpeedEval (cf. Fig. 7(a)).

The TA of Fig. 6(b) computes the difference of both values by the side effect computeDiff(). If the difference diff is bigger than the target-difference, i.e., an error has been detected, the TA triggers a reconfiguration by co4.switchToGPS. This trigger is forwarded by the self-healing automaton of the component SpeedEval (cf. Fig. 7(b)) to the self-healing automaton of component SpeedCtrl (cf. Fig. 8(b)) by co6.switchToGPS. When this message arrives at the self-healing automaton of component TargetSpeed (cf. Fig. 8(b)), the actual reconfiguration is triggered by executeRec().

4.4 Timed Failure Propagation

Having defined the system structure and behavior, we model the system's failure propagation, giving particular attention to its time properties.

We type errors and failures using an extensible failure classification like the one from Fenelon et al. [13]. This allows for the abstraction from concrete errors and failures. This in turn allows for a more precise specification of failure propagation between components. We distinguish the general error and failure classes *service* and *value*. A value error specifies that a value deviates from a correct value, e.g., an erroneous value in the memory of a component. A service error specifies that no value at all is present, e.g., a component crashed.

We use timed failure propagation graphs (TFPGs) to relate an outgoing failure to a set of combinations of errors, incoming failures, and outgoing failures of embedded components. Timing annotations enable the calculation of the propagation times of failures.

For a component specification $sys = (V_{sys}, E_{sys}, s_{sys}, t_{sys})$ over the component types K and the port types P, we first specify the error and failure variables for each component type and then we specify the TFPG for each component type manually. Error and failure variables are named according to the following scheme: $f^d_{k.p,ft}$ and $e_{k,ft}$ for $k \in K$, $p \in P$, $ft \in \{\text{value}, \text{service}\}$, and $d \in \{\text{i}, \text{o}\}$. ft is the set of error and failure classes. i and o specify the direction of failures – i stands for incoming and o for outgoing.

For a configuration, we instantiate error and failure variables when instantiating component types. The notation, described above, holds for component instances and port instances analogously. This instantiation makes all error and failure variable instances unique.

Definition 4.8 (Timed Failure Propagation Graph)
We define $I = \{[\Delta t_{min}, \Delta t_{max}] \mid \Delta t_{min}, \Delta t_{max} \in \mathbb{Q}_{\geq 0},\ \Delta t_{min} \leq \Delta t_{max}\}$ *as the set of* propagation time intervals*,* $\mathcal{E}$ *as the set of* error variables*,* $\mathcal{F}$ *as the set of* failure variables*, and* $O = \{\&, \geq 1\}$ *as the set of* operators*.*

We then define the timed failure propagation graph *(TFPG)* $T = (V, E, f_s, f_t, l, \iota, \eta)$ *as a labeled graph (cf. [11]) over* $(\{\mathcal{E} \cup \mathcal{F} \cup O\}, I)$ *where*

- V *is the set of* nodes*,*
- $E \subseteq V \times V$ *is the set of* edges*,*
- $f_s, f_t : E \to V$ *are the* source *and* target *functions,*
- $l : V \to \{\mathcal{E} \cup \mathcal{F} \cup O\}$ *is the* node labeling *function, and*

– $\iota : E \to I$ *is the* edge labeling *function.*
– $\eta : V \to \{\text{active}, \text{inactive}\}$

We define $\delta^+(v) = |\{e \in E \mid f_s(e) = v\}|$ *as the* out-degree *and* $\delta^-(v) = |\{e \in E \mid f_t(e) = v\}|$ *as the* in-degree *of a node* $v \in V$*. Let* $N_0 = \{v \in V \mid \delta^-(v) = 0\}$*. Then* $\forall v \in V_0 : l(v) \in \mathcal{E}$ *and* $\forall v \in V \setminus V_0 : l(v) \in \mathcal{F} \cup O$ *hold. This means, all nodes with in-degree zero are labeled with error variables. All other nodes are labeled with either a failure variable or a logical operator* $\geq$1 *or* &.

We avoid real numbers as interval bound to enable mapping to time petri nets. This mapping will be introduced in Definition 4.10.

The TFPG of a configuration is built by connecting the TFPGs of all component instances by the connectors. In order to analyze the delays of inter-component communication, the developer has to assign propagation time intervals to the connectors. Figure 10 shows a configuration of the speed control and steering control subsystems from Fig. 2 with the TFPG. Of course, the annotated time values are fictitious.

When a failure propagates through the system over time, we model this fact by the activation of the error and failure variables of the TFPG. In order to express which error or failure variable has been activated, we set the associated node to "active". When the error or failure propagates further through the system, subsequent failure are set to "active".

To model this flow, we add a formal semantics in form of a time petri net [6] to our TFPG. Time petri nets are marked petri nets [36] with a time extension. In this paper, we consider time petri nets in which a transition is labeled with a time interval. Each transition has a clock that is set to 0 each time the transition is enabled. The transition can only fire if this clock has a value that lies within the transition's time interval.

We now give the formal definition of a time petri net as presented in [6]. We assume the same semantics.

Definition 4.9 (Time Petri Net (TPN) [6])
A timed petri net *(TPN)* $\mathcal{T}$ *is a tuple* $(P, T, \cdot(.), (.)\cdot, M_0, (\alpha, \beta))$ *where*

– $P = \{p_1, p_2, ..., p_{|P|}\}$ *is a finite* set of places,
– $T = \{t_1, t_2, ..., t_{|T|}\}$ *is a finite* set of transitions,
– $\cdot(.) : T \mapsto \mathbb{N}^{|P|}$ *is the* backward incidence mapping,
– $(.)\cdot : T \mapsto \mathbb{N}^{|P|}$ *is the* forward incidence mapping,
– $M_0 \in \mathbb{N}_0^{|P|}$ *is the initial marking,*
– $\alpha \in (\mathbb{Q}_{\geq 0})^{|T|}$ *and* $\beta \in (\mathbb{Q}_{\geq 0} \cup \{\infty\})^{|T|}$ *are the* earliest *and the* latest firing time mappings.

We use the morphism defined in Definition 4.10 below to map a TFPG to a TPN. Nodes of the TFPG are mapped to places and edges to transitions. Propagation time intervals become firing time mappings. The minimum propagation time is mapped to the earliest firing time mapping and the maximum propagation time to the latest firing time mapping.

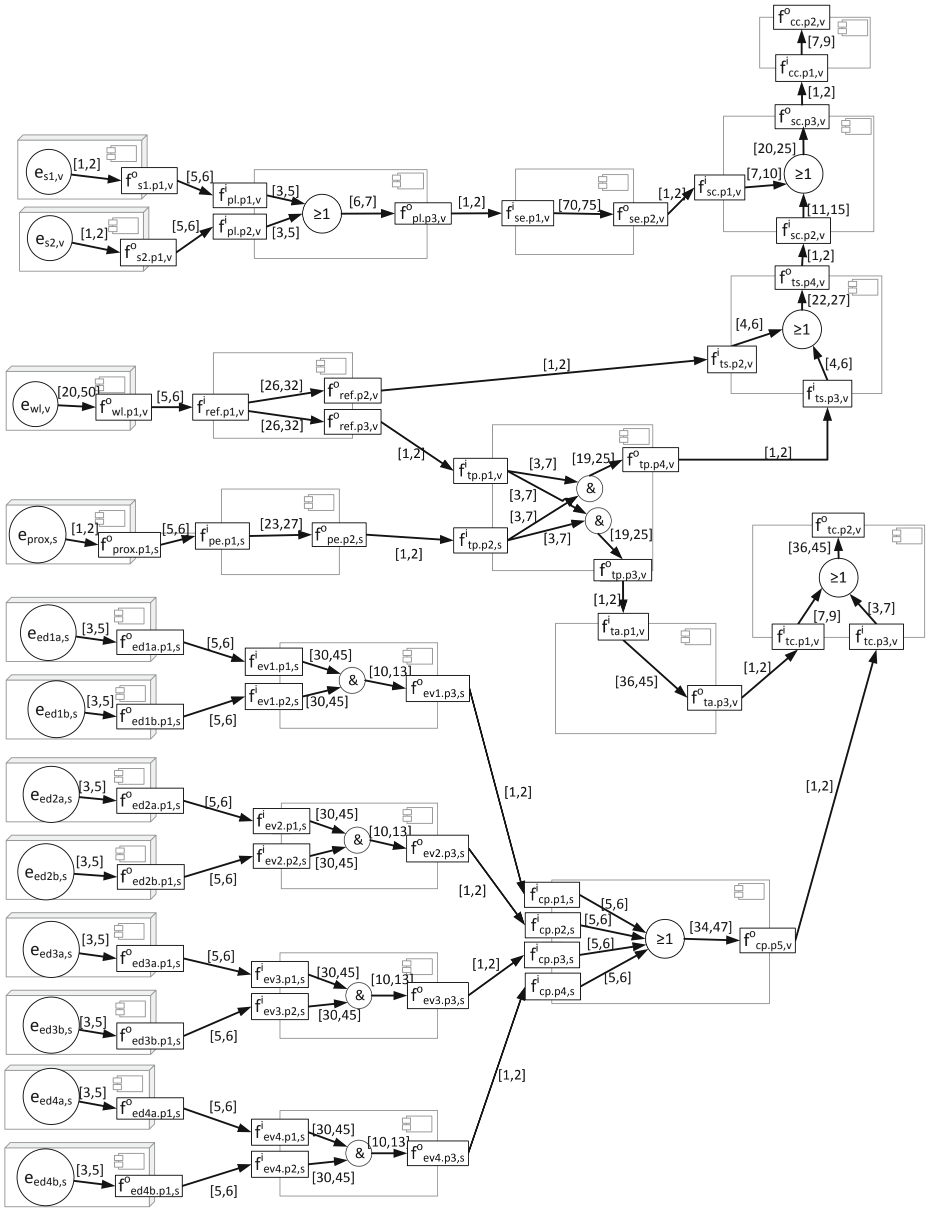


Fig. 10. Configuration with TFPGs

The backward and forward incidence mappings of the TPN are represented by vectors. There exists one vector for backward and forward incidence mapping for each transition, respectively. The size of each vector equals the number of places in the TPN. The entries of the backward incidence mapping specify the number of tokens needed to activate the transition. The entries of the forward incidence mapping specify the number of tokens that move from the transition into the subsequent place.

The backward incidence mapping of a transition t has an entry greater than zero at the index of place p if there exists an edge that originates from p and ends in t. To model the propagation over logical nodes in the TFPG correctly, we need to restrict the propagation via edges leaving AND-nodes in the TFPG. This is achieved by setting the backward incidence mapping of transitions leaving an AND-node to the sum of edges entering the AND-node. The forward incidence mapping of transition t is 1 at the index of place p if p has an incoming edge that originates from t.

Active nodes in the TFPG are mapped to the initial marking. Elements that correspond to active nodes in the TFPG are set to 1. All others are set to 0.

Definition 4.10 (Morphism from TFPG to TPN)
We define a graph morphism $\mu : T \mapsto \mathcal{T}$ from a TFPG $T = (V, E, f_s, f_t, l, \iota, \eta)$ to a TPN $\mathcal{T} = (P, T, ^{\cdot}(.), (.)^{\cdot}, M_0, (\alpha, \beta))$ as a tuple $\mu = (\mu_N, \mu_E, \mu_P)$ where

- $\mu_N : V \to P$ *(bijective),*
- $\mu_E : E \to T$ *(bijective),*
- $\mu_I : I \to (\mathbb{Q}_{\geq 0}^{|T|}, \mathbb{Q}_{\geq 0}^{|T|})$ *(bijective) is the mapping from the set of propagation intervals to the earliest and latest firing time mappings.*

with

- *For all $t \in T$ the backward incidence mapping is described by $^{\cdot}(t) = (v_1, ..., v_{|P|})$, where*

$$v_i = \begin{cases} x & \mu_N^{-1}(p_i) = f_t(\mu_E^{-1}(t)) \\ 0 & else \end{cases}$$

  with

$$x = \begin{cases} \delta^{-}(f_s(\mu_E^{-1}(t))) & l(f_s(\mu_E^{-1}(t))) = \& \\ 1 & else \end{cases}$$

  This means that the backward incidence mapping has only one element that is not zero. This element is the transition following a node labeled with &, and it is set to the in-degree of this &-node.
- *For all $t \in T$ the forward incidence mapping is described by $(t)^{\cdot} = (b_1, ..., b_{|P|})$, where*

$$b_i = \begin{cases} 1 & \mu_N^{-1}(p_i) = f_s(\mu_E^{-1}(t)) \\ 0 & else \end{cases}$$

This means that the forward incidence mapping has only one element that is one.

– $M_0 = (m_1, ..., m_{|P|})$, *where*

$$m_i = \begin{cases} 1 & \eta(\mu^{-1}(p_i)) = \text{active} \\ 0 & else \end{cases}$$

– $\alpha = (\alpha_1, ..., \alpha_{|T|})$ *where* $\alpha_i = \min(\iota(\mu_E^{-1}(t_i)))$

– $\beta = (\beta_1, ..., \beta_{|T|})$ *where* $\beta_i = \max(\iota(\mu_E^{-1}(t_i)))$

Figure 11 shows the TPN of the TFPG of Fig. 10. For better understanding, the labels of the nodes of the TFPG are shown at the corresponding places of the TPN. These labels can be referenced by the reverse application of the morphism and then applying the labeling function of the TFPG: $l(\mu^{-1}(P))$.

In order to make statements about error and failure variables being active or inactive at a point of time, we define the state of a TFPG. As a basis for this, we give the definition of a state of TPNs as used in [6]. A state of a TPN specifies a marking for a period of time. This period of time is represented by a clock zone.

Definition 4.11 (Clock Zone)
Let C be a set of clocks and $B \in 2^{\mathcal{B}(C)}$. A clock zone *$\mathfrak{c}$ is a set of clock interpretations described by conjunction of clock constraints, i.e.*

$$\mathfrak{c} = \bigwedge_{b \in B} b.$$

If C has k clocks, then $\mathfrak{c}$ represents a convex set in the k-dimensional euclidean space. [2]

We use normalized clock zones.

Definition 4.12 (Normalized Clock Zone)
A clock zone c is called normalized*, iff for any given clock a or clock pair a, b it contains either exactly two constraints, or none. The first of these two constrains must use $<, \leq$, while the second must use $>, \geq$. This guarantees a contiguous valid interval for each clock or clock pair.*

We further define the set of all clock zones by $\mathcal{C}$ and the function $\rho : \mathcal{C} \mapsto 2^C$ that returns the set of clocks C of a clock zone $\mathfrak{c}$.

In the remainder we refer to normalize clock zones as clock zones. The definition of a clock zone is now used to define the state of a TPN and a TFPG.

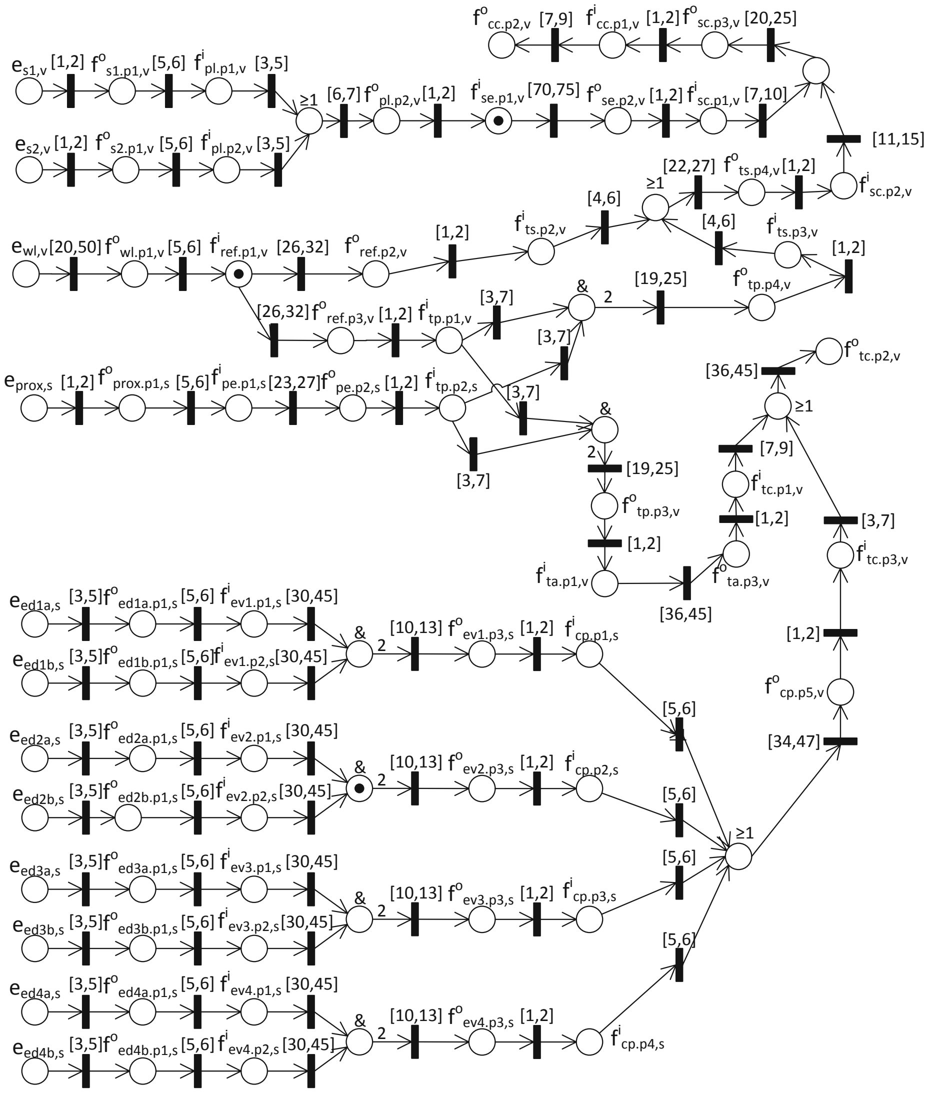


Fig. 11. TPN of the TFPG of Fig. 10 for the state $(A, [45, 61])$

Definition 4.13 (State of a TPN, State of a TFPG)
A state $s = (m, c)$ *of a TPN* $\mathcal{T} = (P, T, ^{\cdot}(.), (.)^{\cdot}, M_0, (\alpha, \beta))$ *is defined by a marking* $m \in \mathbb{N}^{|P|}$ *and a clock zone* c *[6].*

A state $q = (A, \mathcal{C})$ *of a TFPG* $T = (V, E, f_s, f_t, l, \iota)$ *is defined by a set of active error or failure variables* A *and a set of clock zones* $\mathcal{C}$*. Let* $\mathcal{T}$ *be the underlying TPN of* T *and* $\mathcal{T} = \mu(T)$ *the corresponding morphism. Let* $M = (m_1, ..., m_{|P|}) \in \mathbb{N}^{|P|}$ *be a marking in* $\mathcal{T}$*. Then*

$$A = \left(\bigcup_{i=1}^{|P|} \{ l(\mu_N^{-1}(p_i)) \mid m_i > 0 \} \right) \cap (\mathcal{E} \cup \mathcal{F})$$

The set A collects all active error and failure variables represented by a token in the underlying TPN of the TFPG during the time span specified by the clock zone.

A state of a TFPG may represent more than one marking of a TPN. This is because, the clock zone of a TFPG state may cover more than one clock zone of a TPN state.

Figure 11 illustrates the state $(A, [45, 61])$. A is a marking for the clock zone $[45, 61]$. It contains the elements that correspond to the places labeled with $f^i_{se.p1,v}$, $f^i_{ref.p1,v}$, and &. The places initially marked with a token were $e_{s1,v}$, $e_{wl,v}$ and $e_{ed2a,s}$.

Finally, we have to specify hazards. We use boolean formulas to specify the combinations of outgoing failures of a configuration that cause hazards.

Definition 4.14 (Hazard)
A hazard h *is defined by a boolean formula* ψ *over the failure variables* $\mathcal{F}$ *of a component specification sys.*

The boolean formula ψ that defines the hazard is represented by a fault tree. This fault tree connects the outgoing failures of the configuration with the hazard as the top event. Thus, we do not specify propagation time intervals for it. We construct the fault tree by applying manual fault tree analysis [20].

Note that the hazard is defined on the component specification. However, as we analyze configurations, we replace the failure variables in the hazard definition by the instantiated failure variables for each configuration. In the case of multiple instances, e.g., multiple component instances of a component type, we use a disjunction of all instantiated failure variables as a pessimistic abstraction. If this is not appropriate, the developer can manually define the hazard using a fault tree for each configuration individually.

Figure 12 shows the fault tree for the hazard $h_{wrongPos}$ that represents the hazard of a wrong position of the RailCab on the track (cf. Sec. 3) already using the replaced instantiated failure variables. The cause of this hazard is a wrong speed or wrong steering angle due to cc:CurrentCtrl or tc:SteeringCtrl emitting wrong values. This is represented by the failure variables $f^o_{cc.p2,v}$ and

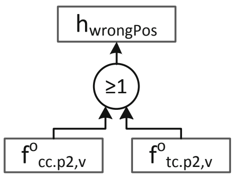


Fig. 12. Fault tree of the Hazard "wrong position"

$f^o_{tc.p2,v}$ in Fig. 12. The hazard is described by the formula $\psi = h_{wrongPos} \Leftrightarrow f^o_{cc.p2,v} \vee f^o_{tc.p2,v}$.

5 Timed Hazard Analysis

Having modeled the failure propagation of our system, we can now perform the timed hazard analysis. Given the configuration, its TFPG, a DGTR, and a matching, the analysis is done in five steps:

Step 1. Determine the minimal cut sets.
Step 2. Determine the reconfiguration delay.
Step 3. Extract the affected subgraph of the configuration's TFPG.
Step 4. Analyze the reachability of failures on the affected subgraph.
Step 5. Analyze the success of the reconfiguration.

In **Step 1**, we compute the minimal cut sets (MCS) of the hazard that is to be reduced by applying our untimed hazard analysis [15]. Steps 2, 3, 4, and 5 are then performed for *each* MCS that the reconfiguration should address. In **Step 2**, we compute the critical time. This is the time span between the detection of the error or failure and the last point in time when the reconfiguration can be executed. This time span corresponds to the time during which a failure will propagate through the system before the reconfiguration is applied. The critical time is determined by means of the reachable behavior of the configuration. In **Step 3**, we extract that part from the system's TFPG that is affected by the reconfiguration, namely the affected subgraph. It is the part of the TFPG that is matched by the DGTR and all paths that lead from error variables into this matching. In **Step 4**, we perform a reachability analysis on the underlying TPN of the affected subgraph. It analyzes how far failures propagate through the affected subgraph during the critical time.

The naive approach of our analysis would be to perform the reachability analysis on the whole TFPG and then apply the changes specified in the DGTR and perform the reachability analysis for an infinite runtime of the system. But reachability analysis on TPNs [6] is equivalent to timed model checking which is exponential on the number of states and clocks [2]. We improve the performance and thus the feasibility of our analysis by performing the reachability analysis on

TPNs only on the part of the configuration that is affected by the reconfiguration. By reducing the number of places of the TPN considered by the reachability analysis we also reduce the number of clocks which have to be analyzed. Thus, we expect an exponentially lower runtime depending on the number of places and clocks not considered by the reachability analysis.

After the reachability analysis, the DGTR is applied to the TFPG. The state of this TFPG contains all errors and failures that are active after the application of the reconfiguration. In **Step 5**, we analyze whether the reconfiguration was successful in preventing the hazard of the MCS. For this, we assign the state of the active error and failure variables resulting from Step 4 to the TFPG of the whole configuration after the reconfiguration. The TFPG represents the situation of errors and failures in the configuration after the reconfiguration. On this TFPG, we perform our untimed hazard analysis. If all resulting MCSs contain at least one inactive error or failure, the reconfiguration has prevented the hazard successfully for the MCS.

One may argue that there are failures that are not addressed by the reconfiguration. These failures may propagate to the hazard without being addressed by the reachability analysis. Step 5 assures that theses failures are detected.

It cannot happen that paths are created by the DGTR that build a "bridge" that lets a failure slip through the self-healing. Hence, it does not occur that the analysis yields that the failure can be stopped though it is not. This is due to the semantics of DGTRs: First, all objects that are to be destroyed, are removed. Then, all objects that are to be created are created.

Cycles in the TFPG can also be handled by our analysis because we map our TFPG to a TPN. The reachability analysis of [6] can handle cycles in TPNs. Further, the untimed analysis of [15] can handle cycles, as well.

In the following, we present the five steps in detail using our running example.

Step 1 – Determine the Minimal Cut sets

The boolean formula of a hazard (cf. Definition 4.14) contains all failure variables that build the top-level failures of the TFPGs of the configuration. A TFPG defines the MCS, i.e., the set of possible causes, for such a top-level failure, [23].

To compute the MCSs, the TFPG (without propagation time intervals) can be transformed into a boolean formula φ. For this, all nodes of the TFPG with out-degree greater than one have to be divided into sub-nodes with out-degree one. Furthermore, all paths consisting solely of failure nodes are replaced by edges. The result is a syntax tree of φ that can be mapped to the corresponding boolean formula φ.

The MCS can be computed from φ using different approaches (exact ones [25,34,42] or heuristics [5,38,39]). As the focus of this paper is not on computing prime implicants of boolean formulas, we refer the interested reader to these publications for details.

It follows a formal definition of cut sets and MCS.

Definition 5.1 (Cut set, Minimal Cut set)
A cut set *s to a failure propagation formula φ of a TFPG T is a conjunction of literals over variables in φ such that $s \to \varphi$ is a tautology. It can be interpreted as an assignment of boolean values to a subset of the variables in φ that guarantees that φ evaluates to true. The set of all such cut sets for φ is denoted $S(\varphi)$.*

A cut set s for φ is a minimal cut set *(MCS) iff no sub-term of s is a cut set for φ. The set of MCSs for φ is denoted $S_m(\varphi)$.*

The MCSs of the failure propagation graph of Fig. 10 are $\{e_{s1,v}\}$, $\{e_{s2,v}\}$, $\{e_{wl,v}\}$, $\{e_{ed1a,s}, e_{ed1b,s}\}$, $\{e_{ed2a,s}, e_{ed2b,s}\}$, $\{e_{ed3a,s}, e_{ed3b,s}\}$, and $\{e_{ed4a,s}, e_{ed4b,s}\}$.

Step 2 – Determine the Reconfiguration Delay

The reconfigurations that we consider represent the reaction to a detected failure in the system. In most cases, the reconfiguration is not executed immediately after the failure has been detected. Typically, the information that a reconfiguration has to be applied must be transmitted to the component at which the reconfiguration is executed. This transmission results in a delay between the detection of an error or failure and the execution of the reconfiguration, namely the critical time. This delay has to be taken into account for the reachability analysis of Step 4, as it is the time span in which errors and failures propagate through the system before the execution of the reconfiguration.

The *critical time* is the sum of the reconfiguration delay and the duration of the DGTR. The *reconfiguration delay* is the time span between the detection of an error or failure and the start of the execution of the DGTR. The duration of the DGTR, i.e., the duration of the execution of the reconfiguration is given by the time interval of the DGTR (cf. Section 4.2).

We determine the reconfiguration delay by analyzing the reachable behavior of the configuration by means of a zone graph [4]. A zone graph contains all runtime states that a network of timed automata (NTA) (cf. Definition 4.7) may visit during its execution. Note that we pessimistically abstract from the real sensor data and the computation on this data. Instead, we consider all possible paths. From the zone graph we extract all paths that contain the message of the detection of the error or failure and the execution of the reconfiguration. The reconfiguration delay is the delay of the path with the maximum delay.

The runtime states of an NTA always consist of the active locations of all contained timed automata and a clock value assignment that assigns a value to each clock. Since clock values are real numbers, there exist infinitely many clock value assignments and, thus, infinitely many states. A reachability analysis that searches the whole state space would not terminate. The problem is tackled by using clock zones as presented in Definition 4.11 that abstract sets of states with the same locations into a single zone [2].

Based on the definition of clock zones, the semantics of an NTA is defined by its zone graph [2,4]. The zone graph contains all possible runtime states of the NTA.

Definition 5.2 (Zone Graph)
Given is an NTA $A_i = (L_i, l_{0_i}, C_i, \Sigma, R, E_i, I_i)$, $i \in \mathbb{N}$. Its reachable state space is given by a zone graph $Z = (S, s_0, \Sigma', T)$ where S is the set of states, s_0 is the initial state, Σ' is the set of transition labels and $T \subseteq S \times \Sigma' \times S$ the set of transitions.

Definition 5.3 (Construction of Zone Graph)
Let A_i and Z be as in Def. 5.2. States are tuples (l, c) where l is a location vector that stores the active location for each automaton and c is a clock zone storing the possible clock interpretations. Let l_i denote the i^{th} element of the location vector l representing the active location of A_i and $l[l'_i/l_i]$ the vector l with l_i being substituted with l'_i. In $s_0 = (l_{init}, c_{init})$, $l_{init,i} = l_{0,i}$ for all A_i and all clocks $c_{0j} \in c_{init}$ have value 0. We define the set of transition labels by $\Sigma' = \{\delta\} \cup \{(i, \tau) | i \in \{1, \ldots, n\}\} \cup \Sigma'_{act}$ with $\Sigma'_{act} = \{((i, a?), (j, a!)) | i \in \{1, \ldots, n\}, j \in \{1, \ldots, n\}, a \in \Sigma\}$. Let $e_j = (l_j, \varphi_j, a_j, r_j, \lambda_j, l'_j) \in E_j$ and $e_m = (l_m, \varphi_m, a_m, r_m, \lambda_m, l'_m) \in E_m$ with $j \neq m$. The transitions of the zone graph Z are defined by the rules:

(1) $(l, c) \xrightarrow{\delta} (l, c + d)$ *if* $c \in I(l)$ *and* $(c + d) \in I(l)$, *where* $I(l) = \bigwedge_{l_i \in l} I(l_i)$ *and* $d \in \mathbb{R}^+$

(2) $(l, c) \xrightarrow{(j,\tau)} (l[l'_j/l_j], c')$ *if* $l_j \xrightarrow{\varphi_j,\tau,\lambda_j} l'_j$, $c' = ((c \wedge \varphi_j)[\lambda_j \mapsto 0]) \wedge I(l[l'_j/l_j])$

(3) $(l, c) \xrightarrow{(j,r)} (l[l'_j/l_j], c')$ *if* $l_j \xrightarrow{\varphi_j,r,\lambda_j} l'_j$, $c' = ((c \wedge \varphi_j)[\lambda_j \mapsto 0]) \wedge I(l[l'_j/l_j])$

(4) $(l, c) \xrightarrow{((j,a?,r),(m,a!,s))} (l[l'_j/l_j][l'_m/l_m], c')$ *where* $r, q \in R$ *if*

  (a) $l_j \xrightarrow{\varphi_j,a?,r,\lambda_j} l'_j$, $l_m \xrightarrow{\varphi_m,a!,q,\lambda_m} l'_m$, *with* $r = \epsilon$ *or* $q = \epsilon$ *and*

  (b) $c' = ((c \wedge \varphi_j \wedge \varphi_m)[\lambda_j \cup \lambda_m \mapsto 0]) \wedge I(l[l'_j/l_j][l'_m/l_m])$. *(cf. [4])*

In the zone graph, four kinds of transitions numbered *(1)*, *(2)*, *(3)*, *(4)* in Definition 5.2 may occur that represent different actions of the NTA. The NTA may *(1)* delay, i.e., no transition fires and time δ passes. If *(2)* the transition is marked with τ, a single timed automaton in the network fires a transition without any further synchronization. If *(3)* the transition carries a side effect r, a single timed automaton in the network fires a transition where the side effect r is executed. If *(4)* two transitions of different time automata exist that use the same action name while one is an output action (!) and the other one is an input action (?), the two transitions may fire synchronously. Only one of these transitions may carry a side effect to avoid interference.

For the computation of the reconfiguration delay, we extract all paths from the zone graph that have the error or failure detection of the respective error or failure and a subsequent reconfiguration as a reaction. We then compute the runtime of these paths with the help of the clock zones of the zone graph paths. The longest runtime over all these paths is the reconfiguration delay.

Since the clocks of the timed automata of the NTA may be reset to 0 by clock resets, the values of the clocks along the path of the zone graph do not reflect the amount of time that was spent on a path directly. Instead, they only measure the time that elapsed after the last reset. Thus, we need to sum up the time that elapsed between two resets along a path of the zone graph. This is done by partitioning the paths into sub paths that start and end at transitions where a reset occurred.

Definition 5.4 (Maximum Runtime Time of a Zone Graph Path) *Consider a path $p \in P$ in the zone graph Z of an NTA with $A_i = (L_i, l_0, C_i, \Sigma, E_i, I_i)$ with $p = t_1, \ldots, t_l$, $t_i = (s_i, \sigma_i, s_{i+1})$, $s_i = (l_i, c_i)$ and $\mathcal{C}_p$ the clock zones of p.*

For a the set of clocks $C_p = \{c \in C | c = \rho(\mathcal{C}_p)\}$ of the path p, we partition p into $t_{11}, \ldots, t_{1n_1}, t_{21}, \ldots, t_{2n_2}, \ldots, t_{m1}, \ldots, t_{mn_m}$ by removing all edges $\{e_{i,j} \in E_i | \lambda_j \neq \varnothing\}$ from p that contain clock resets.

For a partition $p_i = t_{i1}, \ldots, t_{in_i}$, its corresponding clock zones $c_{i1}, \ldots, c_{in_{(i+1)}}$, and for each clock $c_j \in C_p$, we define the following values:

$$t^0_{i,j} = \lim_{c_j \to -\infty}(\exists c_1, \ldots, c_{j-1}, c_{j+1}, \ldots, c_{|C|} \in C_p : c_{i1})$$
$$t^1_{i,j} = \lim_{c_j \to \infty}(\exists c_1, \ldots, c_{j-1}, c_{j+1}, \ldots, c_{|C|} \in C_p : c_{i(n_i+1)})$$

The maximum runtime time of path p is defined by

$$\vartheta_{\max}(p) = \min_{j \in \{1, \ldots, |C_p|\}} \sum_{i=1}^{m} (t^1_{i,j} - t^0_{i,j})$$

For each clock c_j, we compute the minimum value $t^0_{i,j}$ that satisfies the clock zone z_{i1} of the first transition of the partition p_i and the maximum value $t^1_{i,j}$ that satisfies the clock zone $c_{i(n_{i+1})}$ of the last transition of p_i. This clock value must also be a valid clock zone for all other clock zones of the transition. Then, we compute runtime of the path as the sum of the differences of $t^1_{i,j} - t^0_{i,j}$ of all partitions for each clock. Of all sums of all clocks, we take the minimum, because we must choose the clock values such that all clock zones are satisfied. This corresponds to the minimum runtime[8] over all clocks.

In our example, the delay between the reception of the value of the speed sensor s1:SpeedSensor in the timed automaton of Fig. 6(a) and the final trigger of the DGTR by the execution of the side effect in the timed automaton of Fig. 8(b) is $[18, 21]$. The *critical time*, i.e, the time between the arrival of the failure at the component until the completion of the reconfiguration, is calculated by adding the reconfiguration delay and the duration of the DGTR. In our example, this is $[15 + 30, 19 + 42] = [45, 61]$.

[8] We take the greatest value of the minimum values and the smallest value of the maximum values to satisfy all clock zones.

Step 3 – Extract the Affected Subgraph of the Configuration's TFPG

In Step 3, we extract the subgraph of the TFPG that is affected by the reconfiguration and which we will use in Step 4. By only performing the reachability analysis on this subgraph, we improve the feasibility of our analysis.

Definition 5.5 (Affected Subgraph of a TFPG)
Let $w = (G = (V, E, s, t), type : G \to sys)$ be a configuration for a system sys, $P = (L, R, d)$ be a DGTR, matched into G by m. Let further $T = (V_t, E_t, s_t, t_t, \iota, \nu)$ be the TFPG for w. Let the correspondence between T and w be expressed by $\zeta : V_t \to V$.

We then define the set of affected nodes *of G by*

$$V_A = \left(\zeta^{-1} \circ m\right)(L \setminus R)$$

The affected subgraph *G_A of G is induced by the nodes of a set X of paths $x = n_1, \ldots, n_m$ where each $x \in X$ meets the following conditions:*

- $l(v_1) \in \mathcal{E}$ *(path starts in an error)*
- $h(v_m) \notin V_A$ *(path ends at a non-affected node)*
- $h(v_{m-1}) \in V_A$ *(the penultimate node of each path is an affected node)*
- *For all extensions of x,*
  $h(n_i) \notin V_A$ holds for $i > m$. *(path is maximal)*

The affected subgraph consists of the affected nodes, all paths that lead from error nodes to the affected nodes, and the direct successors of the affected nodes. The affected nodes are those node that either appear in the LHS and not in the RHS. They are therefore the nodes which are altered by the DGTR. The affected nodes are needed to judge on the effect of the DGTR on the TFPG. The affected nodes are connected to all nodes of error variables that cause the failures represented by the affected nodes. We can thus analyze whether failures may propagate to the affected nodes. The successors of the affected nodes are needed to analyze whether failures leave the affected part and may still lead to failures which cause the hazard. This information is used in Step 5 to analyze whether the failures that remain in the system may still cause the hazard.

An example that illustrates the affected part of the TFPG of Fig. 10 is shown in Fig. 13. The matching was introduced in Fig. 5. The affected subgraph consists of the two paths $e_{s1,v}$, ..., $\geq$1 and $e_{s2,v}$, ..., $\geq$1.

In order to perform the reachability analysis in Step 4, we have to set the state of the affected part, i.e., the errors and failures that are active. These are the error and failures that have been detected by the system's error detection. In our example, the state of the affected subgraph is set to $(\{f^i_{pl.p1,v}\}, [45, 61])$.

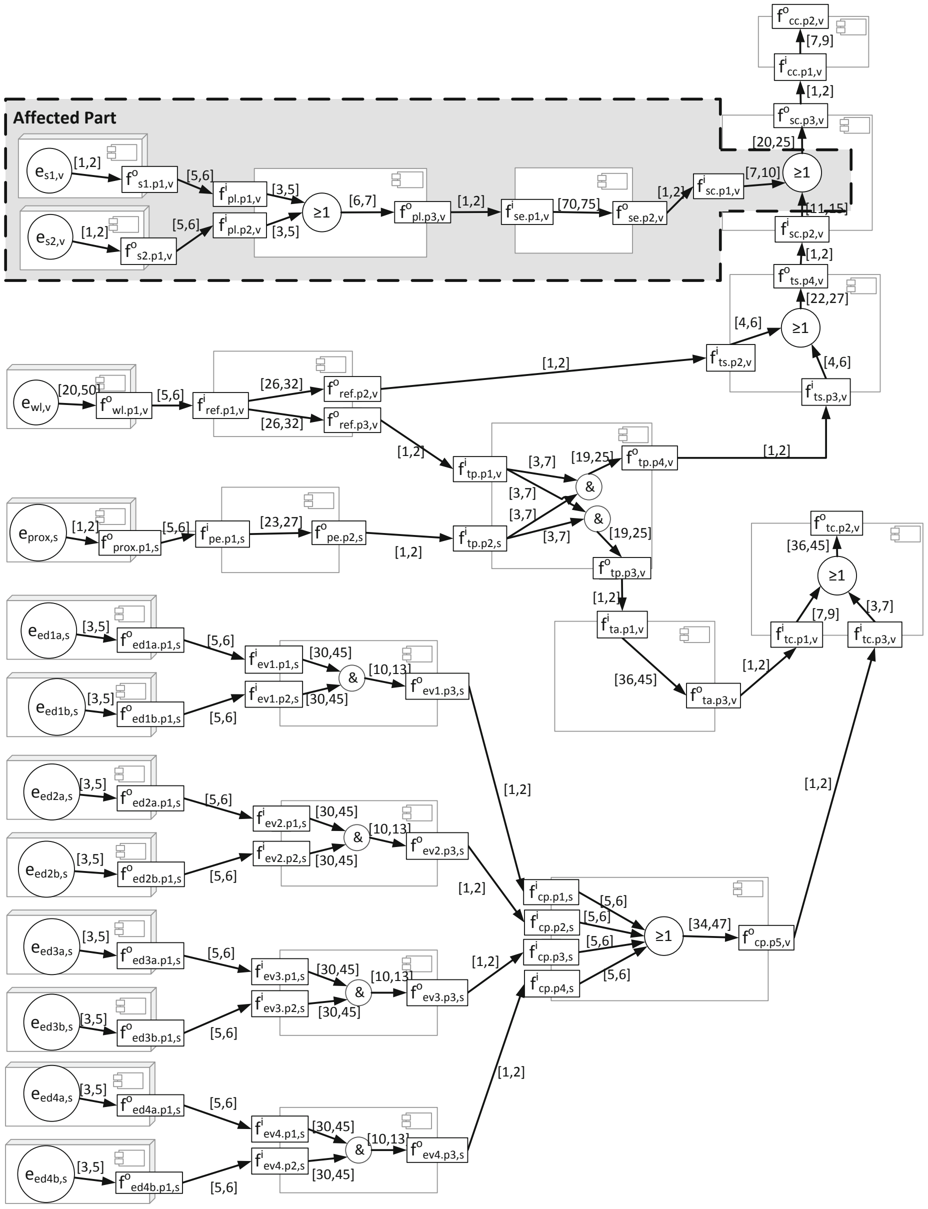


Fig. 13. Affected Part of the Configuration

Step 4 – Analyze Reachability of Failures on Affected TFPG

We now analyze the effect that the DGTR has on the occurrence of a hazard. The DGTR influences the affected subgraph of the TFPG. It may remove error and failure variables or stop errors and failures from propagating further through the system.

We start with the TFPG T of a configuration w and the state of T at the time of the detection of the error or failure. We compute the state (A, z) of the TFPG T for the time interval specified by the reconfiguration delay. The set A then contains all error and failure variables that may be active before the DGTR is executed completely. We take the state at the completion of the DGTR since we do not know exactly, when the individual operations in the DGTR will be performed. By regarding the end of the execution of the DGTR we let the failures propagate the farthest possible path. With this state of the TFPG we apply the DGTR to the TFPG. This means, we change the configuration and at the same time the TFPG according to the DGTR. From this TFPG, we extract the active error and failure variables. These correspond to the errors and failures that remain in the system after the reconfiguration. This computation is made by Algorithm 1.

Algorithm 1 AnalyzeAffectedSubgraph

```
Input: $c = (\overline{K}, \overline{P}, t, \overline{L})$ a configuration,
    $T_A = (V, E, f_s, f_t, l, \iota, \eta)$ the affected subgraph,
    $A \subseteq \mathcal{E} \cup \mathcal{F}$ the active error and failure variables,
    $r = (LHS, RHS, d)$ the DGTR
Output: $A''$ the active error and failure variables after the reconfiguration for the
    lifetime of the system

1: $\mathcal{T} = \mu(G_A)$ with $M_0 = (m_1, ..., m_{|P|})$ with
       $m_i = \begin{cases} 1 & (\mu^{-1}(p_i)) \in A \\ 0 & else \end{cases}$
2: $A' :=$ ComputeTFPGState $(\mathcal{T}, d)$
3: $c \overset{r}{\Rightarrow} \hat{c}$
4: $\hat{T}_A = (\hat{V}, \hat{E}, \hat{f}_s, \hat{f}_t, \hat{l}, \hat{\iota}) :=$ buildAffectedSubgraph($\hat{c}$)
5: $A'' = \{x \in A' \mid x \in \hat{l}(\hat{V})\}$
6: return $A''$
```

Algorithm 1 first creates the TPN of the affected subgraph with the same state as the TFPG (Line 1). On this TPN, it computes the state of the affected subgraph for the duration of the execution of the DGTR (Line 2). This step is explained in further detail in Algorithm 2 below. The algorithm then applies the DGTR on the underlying configuration of the affected subgraph (Line 3) and builds the affected subgraph of the resulting configuration (cf. Step 2) (Line

4). The remaining active error and failure variables of the affected subgraph are collected in A'' (Line 5) and returned (Line 6).

Algorithm 2 computes the state of a TFPG by the corresponding TPN and a clock zone. First, it computes the set $\mathcal{M}$ of reachable markings of the TPN for the clock zone $\mathcal{c}$ using the approach of [6] (Line 2). All nodes of the TFPG of which the corresponding place in the TPN contains a token in at least one marking in $\mathcal{M}$ are gathered in the set A. (Lines 3-9). Then, A holds all error and failure variables that may be active during $\mathcal{c}$.

Algorithm 2 ComputeTFPGState

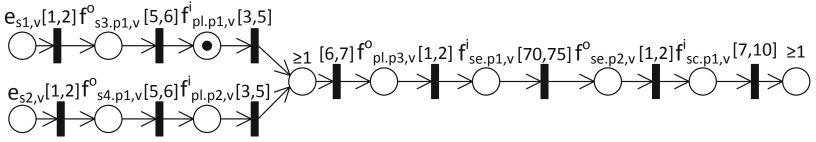
```
Input:  TPN $\mathcal{T}$, clock zone $\mathcal{c}$
Output:  $A$ the set of active error and failure variables for the clock zone $\mathcal{c}$
 1: $A = \varnothing$
 2: $\mathcal{M}$ = getReachableMarkings($\mathcal{T}$, z)
 3: for all $M \in \mathcal{M}$ do
 4:    for $i = 1$ to $|P|$ do
 5:       if $m_i > 0$ then
 6:          $A = A \cup l(\mu^{-1}(p_i))$
 7:       end if
 8:    end for
 9: end for
10: return $A$
```

Figure 14 shows states of the TPN and the TFPG of the affected subgraph during the analysis. Figure 14(a) shows the state of the TPN $\mathcal{T}$ of the affected subgraph of Fig. 13 at the point in time when the failure is detected. Failure variable $f^i_{pl.p1,v}$ has been activated because the failure has been recognized by the component instance pl:Plausibility. Figure 14(b) shows the state of the TPN $\mathcal{T}$ at the end of the duration of the reconfiguration but before the application of the DGTR. During the critical time $[45, 61]$ of the DGTR shown in Fig. 5 the failure variables $f^i_{pl.p1,v}$, $f^o_{pl.p3,v}$, and $f^i_{se.p1,v}$ are reachable in the TPN. They may thus be activated in the real system. Figure 14(c) shows the TFPG marked with the active error and failure variables of the set A' that correspond to the marked places in the TPN of Fig. 14(b). This is the state of the TFPG at the end of the reconfiguration time.

Step 5 – Analyze Success of Reconfiguration

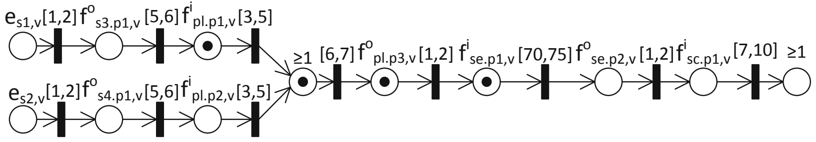
In the last step, we analyze whether the remaining active error and failure variables can still cause the hazard that was to be reduced. For this, we compute the MCSs of the TFPG T' of the configuration after the reconfiguration and check if these MCSs only contain active error or failure variables.

We take the set A'' of active error and failure variables that remain in the system after the reconfiguration, i.e., that were the result of Algorithm 1.



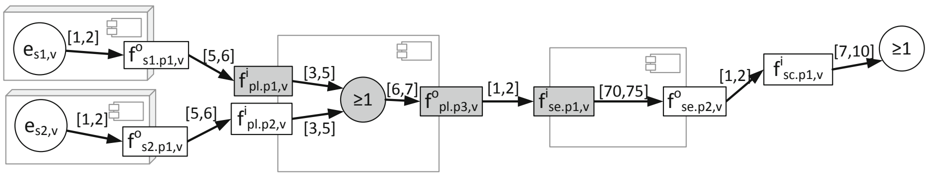
$e_{s1,v}$ [1,2] $f^o_{s3.p1,v}$ [5,6] $f^i_{pl.p1,v}$ [3,5]
$e_{s2,v}$ [1,2] $f^o_{s4.p1,v}$ [5,6] $f^i_{pl.p2,v}$ [3,5]
≥1 [6,7] $f^o_{pl.p3,v}$ [1,2] $f^i_{se.p1,v}$ [70,75] $f^o_{se.p2,v}$ [1,2] $f^i_{sc.p1,v}$ [7,10] ≥1

(a) Marking of the TPN at the Start of the Reconfiguration



$e_{s1,v}$ [1,2] $f^o_{s3.p1,v}$ [5,6] $f^i_{pl.p1,v}$ [3,5]
$e_{s2,v}$ [1,2] $f^o_{s4.p1,v}$ [5,6] $f^i_{pl.p2,v}$ [3,5]
≥1 [6,7] $f^o_{pl.p3,v}$ [1,2] $f^i_{se.p1,v}$ [70,75] $f^o_{se.p2,v}$ [1,2] $f^i_{sc.p1,v}$ [7,10] ≥1

(b) Marking of the TPN at the End of the Reconfiguration



(c) State of the Affected Subgraph for the Duration of the DGTR

Fig. 14. TPN and TFPG during Reconfiguration

We assign A'', i.e., the active error and failure variables to the TFPG T' of the whole system after the application of the reconfiguration.

Before applying the untimed hazard analysis [15], we modify T' such that active failure nodes become basic events. Basic events are nodes that have an in-degree of zero. Thus, according to Definition 4.8, all nodes labeled with error variables are basic events. In our untimed hazard analysis, MCS only contain such basic events. But T' may contain active failure variables of which *none* of the causing error variables are active. Consequently, the MCSs resulting from this TFPG may contain *in*active error variables resulting in active failure variables. A failure that leads to the hazard would not be recognized.

In order to make failure nodes basic events, we delete the incoming edges of all failure nodes. In this way only the failure node with the shortest distance to the outgoing failure that is part of the hazard specification (cf. Definition 4.14) remains connected to this outgoing failure.

Figure 15 shows the system's TFPG after the execution of the DGTR. Due to the application of the DGTR, the edge between $f^o_{se.p2,c}$ and $f^i_{sc.p1,v}$ has been removed. Further, the incoming edges of $f^i_{pl.p1,v}$, $f^o_{pl.p3,v}$, and $f^i_{se.p1,v}$ have been removed because these failure variables are active.

We compute the MCSs of this modified TFPG using our untimed hazard analysis. Then, we check whether any of the resulting MCSs only contains active error or failure variables. If there is no such MCS, the hazard cannot occur, because in each MCS all error and failure variables have to be active to cause

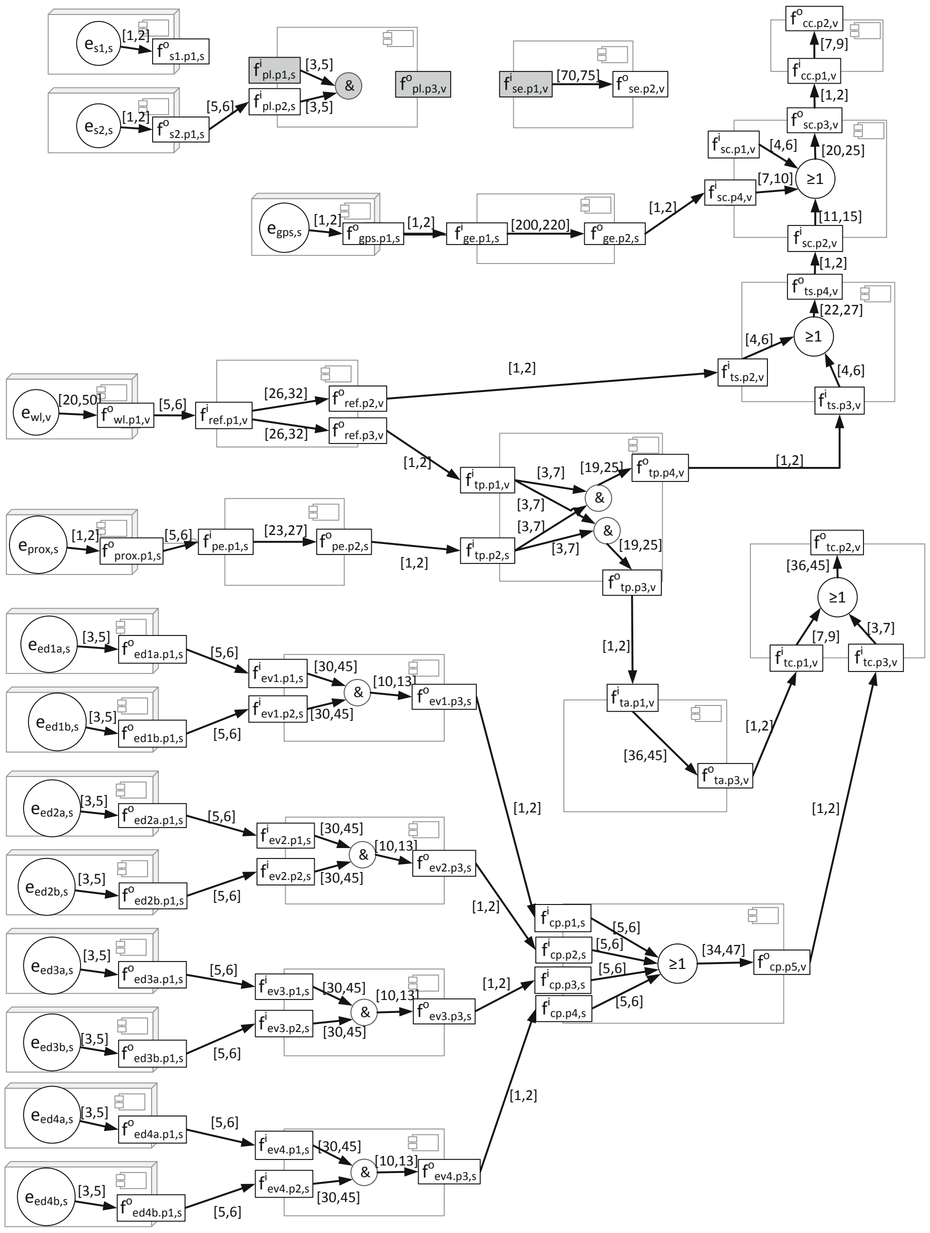


Fig. 15. Reduced TFPG of Step 5

the hazard. If we find an MCS that only consists of active error and failure variables, the hazard cannot be prevented by the applied reconfiguration.

For the TFPG of Fig. 15, we get the MCSs $\{f^i_{sc.p1,v}\}$, $\{e_{gps,s}\}$, $\{e_{wl,w}\}$, $\{e_{ed1a,s}, e_{ed1b,s}\}$, $\{e_{ed2a,s}, e_{ed2b,s}\}$, $\{e_{ed3a,s}, e_{ed3b,s}\}$, and $\{e_{ed4a,s}, e_{ed4b,s}\}$. The set of active failure variables is $\{f^i_{pl.p1,v}, f^o_{pl.p3,v}, f^i_{se.p1,v}\}$. None of the resulting MCSs contains these variables. Thus, the hazard has been reduced successfully.

The presented approach handles multiple errors as well as single errors. This is guaranteed by applying Step 2 to 5 to *cut sets* instead of single errors. These cut sets contain either multiple or single errors depending on which errors are causing the hazard.

In our timed hazard analysis errors and failures are analyzed differently depending on whether they are affected by the reconfiguration or not. Errors and failures that are handled by the reconfiguration are considered in Steps 3 and 4 because we need to know how far they may propagate during the critical time. Step 5 addresses all errors and failures. Here we want to know whether errors and failures that remain in the system after reconfiguration still may lead to the hazard.

6 Evaluation

The evaluation results we present stem from the application of timed hazard analysis on a generated component structure. We aim at analyzing the impact of the optimization of the analysis. For this we apply the timed hazard analysis two times on the same model. First, the reachability analysis is only applied on the affected part. Second, we apply it on the full TFPG. We particularly focus on the execution time with respect to the size of the TFPG. We omit the part of the reachability analysis of Step 2 (cf. Sec. 5).

Our generated configurations have a structure as illustrated in Fig. 16. The TFPG consists of the two paths $e_1, \ldots, f_{j3}$ and $e_2, \ldots, f_{j3}$. The reconfiguration deletes failure variables f_n and f_{j1} and the edges between them (highlighted in red). Thus, the affected part is built by the path $e_1, \ldots, f_{j1}$. For evaluation, we extended the length of both paths equally by one component instance per evaluation step. We repeated the experiment with different numbers of paths in the affected part as sketched in Fig. 16.

The evaluation experiments were executed on a SuSE 11.4 machine with 72 GB RAM and 8 64-bit CPUs with 2.2 GHz clock and 8 MB cache memory.

Figures 17(a) and 17(b) show the runtimes[9] for component structures with an affected part containing 1 and 2 paths, respectively. Note, that the vertical axis has a logarithmic scale. The green lines with triangles show the runtimes of the optimized analysis with respect to path length. The red lines with diamonds show the same fact for the reachability analysis applied on the whole system. In both diagrams, the time saving is significant. Thus, the optimization increases the feasibility of the approach significantly.

[9] Note that only a single CPU was used for each experiment.

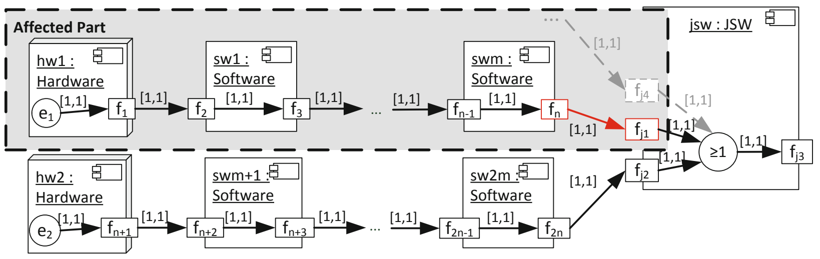


Fig. 16. Configuration Used for Evaluation

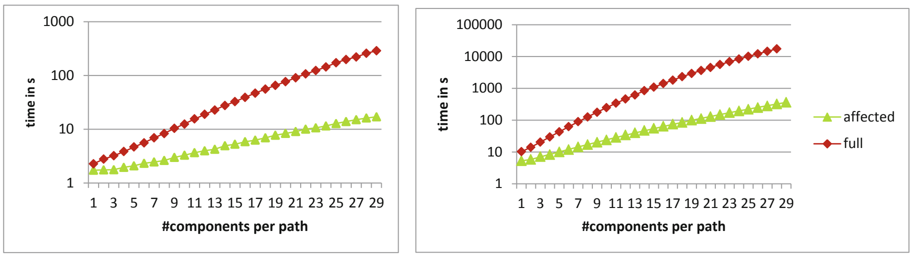


Fig. 17. Evaluation Results

With our approach, we are able to analyze the example with a length of 29 components and 2 paths in the affected part in 321 seconds using our optimization. The same analysis takes 17401 seconds without the optimization.

7 Related Work

There already exist methods for the hazard analysis of technical systems [1,8,17,21,24,28,29,43].

Approaches that apply model checking for hazard analysis are the works of Güdemann et al. [28] and Colvin et al. [8]. Due to state explosion, the method of [28] is only applicable with bounded model checking and can thus not show the absence of flaws. In contrast, our approach can handle the whole system, as our failure propagation graph are an abstraction of the system behavior.

The approaches of Palshikar [29] and Walker et al. [43] take the temporal ordering of errors into account. In contrast to our approach, the analysis yields minimal cut sequences, i.e., sequences of events that are necessary to cause an event, but no concrete time values.

Approaches that consider concrete time values are the works of Colvin et al. [8], Abdelwahed et al. [1], Güdemann et al. [28], Grunske et al. [17] and

Magott et al. [24]. In [24] fault trees are used that encode temporal properties in gates. They compute temporal dependencies between errors whereas we are computing propagation times. The approach of [8] performs a timed Failure Modes and Effects Analysis on timed Behavior Trees. In [17] State Event Fault Trees (SEFT) – Fault Trees combined with state charts – are used. The SEFTs are transformed to Deterministic and Stochastic Petri Nets which contain probability distributions over time for their transitions. The method of [28] is a formal approach for the fault tree analysis by model checking. All these methods allow statements about temporal properties of hazards. Though, they do not analyze propagation times.

The approaches of [29], [43], [8], and [24] do not consider reconfiguration. Approaches that analyze reconfigurable systems are the approaches of [1], [28], and [17]. But they all do not take complex structural rule-based reconfiguration into account. The approach of [17] analyzes each configuration individually, but not the changes themselves. In [28] reconfiguration is specified by state changes and invariants that the system has to satisfy. The method of [1] uses mode variables on their models to switch part of their models on and off. Unlike all these approaches we specify graph transformation rules that define the reconfigurations that can be executed. These reconfiguration rules enable us to analyze failure propagation during the process of reconfiguration.

Another field related to our approach is model-based diagnosis [9, 35]. These approaches aim at diagnosing the root failures of individual components based on observations of the system's faulty behavior. The mentioned approaches mainly use first-order logic to model the behavior of the system. They use the model to derive the minimal set of components whose failures explain an observation. While the employed model shares similarities with our timed failure propagation models, we assume that we are able to directly observe the root failures and we use the failure propagation models to check that a hazard is successfully reduced by structural reconfiguration in reaction to the detection of the root failures. However, these approaches may complement ours with respect to failure diagnosis.

Model-based diagnosis approaches have been enhanced by also supporting repairs to recover from failures in [45]. This approach is restricted to restarting of individual components while we employ explicit reconfiguration rules which support arbitrary structural reconfigurations. Additionally, this approach does not consider time which is important for embedded safety-critical systems.

8 Conclusion and Future Work

In this paper, we presented an approach for the timed hazard analysis of systems that reconfigure in order to react to errors in the system, so called self-healing systems. With our analysis, the system developer is able to analyze whether the system's reaction to an error in terms of reconfiguration is fast enough to prevent a hazard.

In order to analyze the propagation of failures during reconfiguration, we extended our failure propagation models by propagation times and define a formal semantics in terms of Time Petri Nets. This allows for analyzing the reachability of failures in the failure propagation model over time. Timed hazard analysis becomes applicable by performing the reachability analysis only on that part of the system that is affected by the reconfiguration. Further, our approach takes the delay between the detection of a failure and the reconfiguration into account by analyzing the system's behavior in terms of timed automata [4].

The presented approach analyzes the timed automata that model the system behavior to compute the delay between failure detection and reconfiguration. Currently, this analysis requires the computation of the reachable state space of the complete system which may suffer from state space explosion. In the future, we will work on decomposing the system to address this problem exploiting existing compositional verification approaches.

The manual specification of TFPGs is error-prone because software of self-healing systems and thus their software models are very complex. Therefore, we want to develop an automatic generation of timed failure propagation graphs from timed automata that specify the behavior of our systems.

We also plan to add probability distributions over time to our propagation time intervals, as the number and the width of these interval affects the uncertainty of the analysis. With the help of probability distributions we can estimate, e.g., the most probable propagation times of failures.

Acknowledgments. This work was developed in the course of the Special Research Initiative 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft. We thank Anas Anis and Sebastian Lehrig for implementing the tools and the evaluation. We thank Christian Berger for proof reading.

References

1. Abdelwahed, S., Karsai, G., Nagabhushan, M., Ofsthun, S.C.: Practical implementation of diagnosis systems using timed failure propagation graph models. IEEE Transactions on Instrumentation and Measurement 58(2), 240–247 (2009)
2. Alur, R.: Timed Automata. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 8–22. Springer, Heidelberg (1999)
3. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secur. Comput. 1(1), 11–33 (2004)
4. Bengtsson, J., Yi, W.: Timed Automata: Semantics, Algorithms and Tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)
5. Brayton, R.K., Sangiovanni-Vincentelli, A.L., McMullen, C.T., Hachtel, G.D.: Logic Minimization Algorithms for VLSI Synthesis (1984)
6. Cassez, F., Roux, O.-H.: Structural translation from time petri nets to timed automata. Electron. Notes Theor. Comput. Sci. 128, 145–160 (2005)

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.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
8. Colvin, R., Grunske, L., Winter, K.: Timed behavior trees for failure mode and effects analysis of time-critical systems. J. Syst. Softw. 81, 2163–2182 (2008)
9. de Kler, J., Mackworth, A.K., Reiter, R.: Characterizing dianosis and systems. Artifical Intelligence 56 (1992)
10. Eckardt, T., Heinzemann, C., Henkler, S., Hirsch, M., Priesterjahn, C., Schäfer, W.: Modeling and verifying dynamic communication structures based on graph transformations. In: Computer Science – Research and Development, Springer (2011)
11. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. Springer (2006)
12. Fenelon, P., McDermid, J.A.: An integrated tool set for software safety analysis. Journal of Systems and Software 21(3), 279–290 (1993)
13. Fenelon, P., McDermid, J.A., Nicolson, M., Pumfrey, D.J.: Towards integrated safety analysis and design. ACM SIGAPP Applied Computing Review 2(1), 21–32 (1994)
14. Giese, H., Tichy, M.: Component-based hazard analysis: Optimal designs, product lines, and online-reconfiguration. In: Proc. of the 25th International Conference on Computer Safety, Security and Reliability, Gdansk, Poland (2006)
15. Giese, H., Tichy, M., Schilling, D.: Compositional hazard analysis of uml components and deployment models. In: Proc. of the 23rd SAFECOMP, Potsdam, Germany (2004)
16. Grunske, L.: Annotation of Component Specifications with Modular Analysis Models for Safety Properties. In: Overhage, S., Turowski, K. (eds.) Proc. of the 1st Int. Workshop on Component Engineering Methodology, Erfurt, Germany (2003)
17. Grunske, L., Kaiser, B., Papadopoulos, Y.: Model-Driven Safety Evaluation with State-Event-Based Component Failure Annotations. In: Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Ren, X.-M., Wallnau, K. (eds.) CBSE 2005. LNCS, vol. 3489, pp. 33–48. Springer, Heidelberg (2005)
18. Güdemann, M., Ortmeier, F.: Probabilistic model-based safety analysis. In: Pierro, A.D., Norman, G. (eds.) Proceedings Eighth Workshop on Quantitative Aspects of Programming Languages. EPTCS, vol. 28, pp. 114–128 (2010)
19. Henke, C., Tichy, M., Böcker, J., Schäfer, W.: Organization and control of autonomous railway convoys. In: Proceedings of the 9th International Symposium on Advanced Vehicle Control, Kobe, Japan (October 2008)
20. International Electrotechnical Commission, Geneva, Switzerland. International Standard IEC 61025. Fault Tree Analysis, FTA (1990)
21. Kaiser, B., Gramlich, C., Förster, M.: State/event fault trees–A safety analysis model for software-controlled systems. Reliability Engineering & System Safety 92(11), 1521–1537 (2007)
22. Kaiser, B., Liggesmeyer, P., Maeckel, O.: A New Component Concept for Fault Trees. In: Proceedings of the 8th National Workshop on Safety Critical Systems and Software (SCS 2003), Canberra, Australia. Research and Practice in Information Technology, vol. 33 (October 9-10, 2003)

23. Leveson, N.G.: Safeware: System Safety and Computers. ACM (1995)
24. Magott, J., Skrobanek, P.: A method of analysis of fault trees with time dependencies. In: Koornneef, F., van der Meulen, M.J.P. (eds.) SAFECOMP 2000. LNCS, vol. 1943, pp. 176–186. Springer, Heidelberg (2000)
25. McCluskey, E.J.: Minimization of Boolean Functions. Bell System Technical Journal 35 (1956)
26. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing adaptive software. Computer 37(7), 56–64 (2004)
27. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: ICSE, pp. 177–186 (1998)
28. Ortmeier, F., Reif, W., Schellhorn, G.: Deductive cause-consequence analysis. In: Proceedings of the 16th IFAC World Congress (2006)
29. Palshikar, G.K.: Temporal fault trees. Information and Software Technology 44(3), 137–150 (2002)
30. Papadopoulos, Y., McDermid, J.A., Sasse, R., Heiner, G.: Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. Int. Journal of Reliability Engineering and System Safety 71(3), 229–247 (2001)
31. Priesterjahn, C.: Hazard analysis of self-optimizing mechatronic systems. In: Proc. of the Doctoral Symposium of the 7th ESEC-FSE, Amsterdam, The Netherlands (2009)
32. Priesterjahn, C., Heinzemann, C., Schäfer, W.: From timed automata to timed failure propagation graphs. Technical Report tr-ri-12-325, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn (2012), `http://www.cs.uni-paderborn.de/uploads/tx_sibibtex/PHS12_ag.pdf`
33. Priesterjahn, C., Steenken, D., Tichy, M.: Component-based timed hazard analysis of self-healing systems. In: Proceedings of the 8th Workshop on Assurances for self-Adaptive Systems, ASAS 2011, pp. 34–43. ACM, New York (2011)
34. Rauzy, A., Dutuit, Y.: Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within Aralia. Reliability Engineering & System Safety 58(2), 127–144 (1997)
35. Reiter, R.: A theory of diagnosis from first principles. Artifical Intelligence 32(1), 57–95 (1987)
36. Reutenauer, C.: The mathematics of Petri nets. Prentice-Hall, Inc., Upper Saddle River (1990)
37. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations. Foundations, vol. 1. World Scientific (1997)
38. Rudell, R.L.: Multiple-Value Logic Minimization for PLA Synthesis. Technical Report M86/65, University of California at Berkeley, USA (June 1984)
39. Seda, M.: Heuristic Set-Covering-Based Postprocessing for Improving the Quine-McCluskey Method. International Journal of Computational Intelligence (IJCI) 4(2), 139–143 (2008)
40. Storey, N.: Safety-Critical Computer Systems. Addison Wesley (1996)
41. Tichy, M., Henkler, S., Holtmann, J., Oberthür, S.: Component story diagrams: A transformation language for component structures in mechatronic systems. In: Postproc. of the 4th Workshop OMER, Paderborn, Germany. HNI Verlagsschriftenreihe (2008)

42. van Orman Quine, W.: A Way to Simplify Truth Functions. The American Mathematical Monthly 62 (1955)
43. Walker, M., Bottaci, L., Papadopoulos, Y.: Compositional Temporal Fault Tree Analysis. In: Saglietti, F., Oster, N. (eds.) SAFECOMP 2007. LNCS, vol. 4680, pp. 106–119. Springer, Heidelberg (2007)
44. Wallace, M.: Modular architectural representation and analysis of fault propagation and transformation. Electronic Notes in Theoretical Computer Science 141, 53–71 (2005)
45. Weber, J., Wotawa, F.: Diagnosis and repair of dependent failures in the control system of a mobile autonomous robot. Appl. Intell. 36(3), 511–528 (2012)

Model-Driven Development of Safe Self-optimizing Mechatronic Systems with MechatronicUML[*]

Holger Giese[1] and Wilhelm Schäfer[2]

[1] Hasso Plattner Institute for Software Systems Engineering at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam, Germany
holger.giese@hpi.uni-potsdam.de

[2] Heinz Nixdorf Institute at the University of Paderborn
Zukunftsmeile 1, D-33102 Paderborn, Germany
wilhelm@uni-paderborn.de

Abstract. Software is expected to become the dominant driver for innovation for the next generation of advanced distributed embedded real-time systems (advanced mechatronic systems). Software will build communities of autonomous agents at runtime which exploit local and global networking to enhance and optimize their functionality leading to self-adaptation or self-optimization. However, current development techniques are not capable of providing the safety guarantees required for this class of systems. Our approach, MechatronicUML, addresses the outlined challenge by proposing a coherent and integrated model-driven development approach which supports the modeling and verification of safety guarantees for systems with reconfiguration of software components at runtime. Modeling is based on a syntactically and semantically rigorously defined and partially refined subset of UML. Verification is based on a special type of decomposition and compositional model checking to make it scalable.

1 Introduction

Software has become an intrinsic part of complex distributed embedded real-time systems, also referred to as mechatronic systems. In many cases these systems are used in a safety-critical environment and implement themselves as so-called safety-critical applications. Consequently, the development of software controlling these systems has to undergo a rigorous process including the prevention of faults, employing adequate and well-founded modeling concepts, and the verification of crucial safety properties in order to detect critical faults.

[*] This work was developed partially in the course of the Special Research Initiative 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – at the University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

The outlined requirements are also valid for the next generation of advanced mechatronic systems. These systems are expected to behave more intelligently than today's systems by building communities of autonomous agents which exploit local and global networking to enhance their functionality [1] also named cyber-physical systems. Such mechatronic systems will employ different forms of reconfiguration to enable self-adaptation [2] often particularly targeting self-optimization in this domain. These forms of reconfiguration include complex coordination protocols which require execution in real-time, reconfiguration of control algorithms as well as components, and the coordination of the agents at runtime to adjust their behavior to the changing system goals. However, the available development techniques cannot handle systems with these advanced forms of reconfiguration.

We address this challenge with the model-driven MechatronicUML (mUML) development approach, which combines domain specific modeling and refinement techniques with verification based on compositional model checking. The approach suggests modeling the software by using a refined UML component model, including the detailed definition of ports, connectors, and patterns/collaborations. We further refine the component model to define proper integration between discrete and continuous control so that the reconfiguration of hierarchical component systems can be described in a modular way. Compositional model checking is based on a domain specific decomposition of the system specification into individually checkable components and patterns/collaborations based on a common predefined architectural model. The basis for formal verification, a formal semantics for the concepts taken from UML, is given in [3].

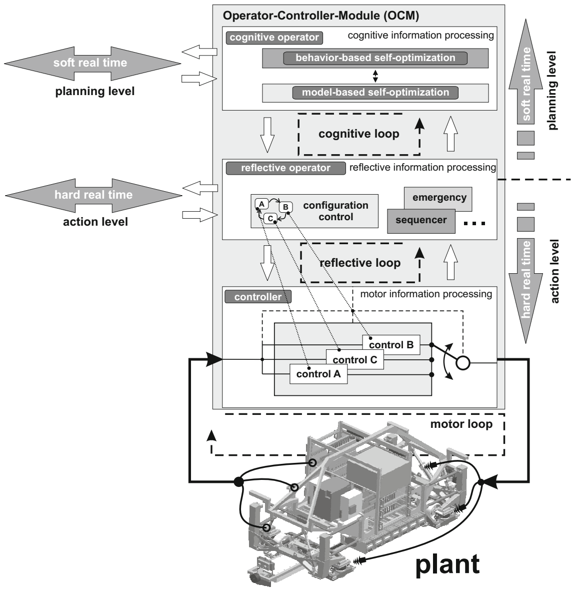
The paper contains the following new contributions: (1) A comprehensive overview of the rationale behind the mUML MDD approach combining modeling and verification, previously covered only partially in [4, 5]. (2) A rigorous integration of the previously only independently outlined mUML modeling concepts for modeling hierarchies of reconfigurable components with hybrid behavior [6–10] and the real-time coordination of mechatronic agents [6, 7]. (3) Finally, an approach for the overall verification based on the modular verification concepts for hierarchies of reconfigurable components with hybrid behavior [6–10] and the compositional verification of the real-time coordination of mechatronic agents [6, 7] which has not been covered before.

The structure of this paper is as follows: The next section provides an overview of the mUML approach and introduces our running example, explains the underlying general architectural model, outlines how self-optimization takes place in this architecture, and provides an overview of the modeling and verification of mUML models. Section 3 introduces the modeling concepts in the form of a component model permitting specific structural reconfiguration as well as behavior specification. The concepts for modeling real-time coordination of the components are also outlined. In Section 4 the local safety criteria which have to be verified are defined and it is shown that their composition ensures global safety. In Section 5 we review existing work in the field and compare it with our approach. The last section concludes the paper.

2 The Approach

As a specific example of an advanced mechatronic system, we use the Paderborn-based RailCab research project (http://www-nbp.upb.de/en), which aims at combining a passive track system with intelligent shuttles operating individually and making independent and decentralized operational decisions. The project is funded by a number of German research organizations. A test track has been built to the scale of 1:2.5 so that the project's concetps can be tested in real operation and not just on paper (cf. Fig. 1(a)).

(a) Test track



(b) OCM architecture and its elements

Fig. 1. The test track of the RailCab project and the OCM architecture

The RailCab project aims to provide the comfort of individual transport concerning scheduling and on-demand availability of transportation as well as individually equipped cars together with the cost and resource effectiveness of public transport. The modular railway system combines sophisticated undercarriages with the advantages of new actuation techniques as employed in the Transrapid (http://www.transrapid.de) to increase passenger comfort while still enabling high speed transportation and (re)using the existing railway tracks.

One particular goal of the project is to reduce the energy consumption due to air resistance by coordinating the autonomously operating shuttles in such a way that they build convoys whenever possible. Such convoys are built on-demand and the shuttles travel only a few centimeters apart from each other (up to

0.5m) so that a high reduction of energy consumption is achieved. Consequently, coordination between the speed control units of the shuttles becomes a safety-critical aspect and results in a number of hard real-time constraints which have to be addressed when designing the control software of the shuttles and the real-time coordination between the shuttles.

2.1 The General Architectural Model

In order to build such a complex software system, the MUML approach follows a general local architectural model of a system component for self-optimizing mechatronic systems given by the Operator-Controller-Module (OCM) as depicted in Fig. 1(b) (cf. [11]).

The OCM reflects the strict hierarchical construction of autonomous mechatronic systems, including the hardware components structured, into three levels: (1) On the lowest level of the OCM is the controller (C) including an arbitrary number of alternative control strategies (also called modes from an external perspective). Within the OCM's innermost loop, the currently active control strategy processes measurements obtained via sensors and produces control signals for the actuators. As it directly affects the plant, it is called a motor loop. The software processing is necessarily quasi-continuous and includes smooth switching between the alternative control strategies described by some form of differential equations or difference equations. (2) The controller is controlled by the reflective operator (RO), in which monitoring and controlling routines are executed. The reflective operator operates in a predominantly event-oriented manner and thus includes a control automaton with a number of discrete control states and transitions between them. It does not access the actuators of the system directly, but may modify the controller and initiate the switch between different control modes and its related strategies. Furthermore, it serves as the connecting element to the cognitive level of the OCM. (3) The topmost level of the OCM is called the cognitive operator (CO). On this level, the system can gather information concerning itself and its environment and use it for the improvement of its own behavior. (i.e. possibly complex, time-consuming computations for long-range planning.)

The distinction between the reflective and cognitive operator clearly decouples control under hard real-time constraints from long-range planning and the resulting input for self-optimization. In general the OCM-hierarchy defines a strictly hierarchical control flow. Each level tries to execute control as much as possible locally, but reconfiguration of components is decided on the next higher level similar to the reference architecture for adaptive and self-managed systems suggested in [12].

To also describe the overall architecture, the OCM hierarchy can be nested, where each nesting level may include an OCM. However, these levels do not include the controller part. Controllers, which implement the continuous part of the software, usually exist only on the lowest level of a nested OCM hierarchy. As an example, consider the above mentioned shuttles of the RailCab project. The architecture is defined by OCMs with their reflective operators and

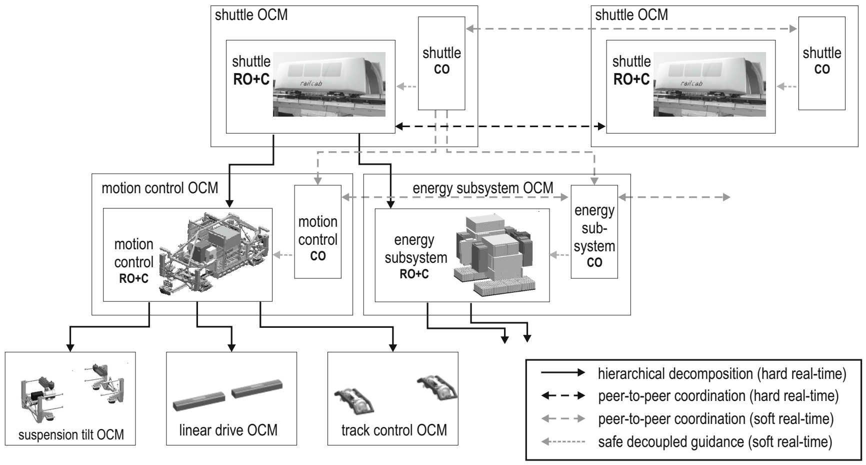


Fig. 2. The hierarchy of OCMs of a shuttle and its connections to other shuttles

the controllers as depicted in Fig. 2. A shuttle consists of components like the suspension/tilt module, the engine, the tracking module etc. which in turn are defined by OCMs.

As a complete mechatronic system usually consists of several concurrently running components, a further possibility for communication between components besides the strict hierarchical control flow exists. Top-level OCMs of several nested hierarchies, which usually represent a major system component, may act as freely interacting software agents in the overall architecture in addition to the strict hierarchies. This means that agents exchange information and collaborate in a peer-to-peer manner but that no central control is defined anymore. As examples of such major system components consider the different shuttles, stations and possibly job brokers involved with the RailCab project. These agents interact with each other in form of collaborations with well-defined role interfaces. In principle, the controllers of different agents can interact with each other, as well as the reflective operators and the cognitive operators, each on their corresponding levels. In any case their interaction is limited to a peer-to-peer style with individual messages rather than centralized, broadcasted messages.

2.2 Self-adaptation and Self-optimization

Self-optimization by means of self-adaptation can be realized in rather different forms in the outlined general architectural model depending on the specific self-optimization goals and the impact the different elements have concerning the characteristics that should be optimized.

The most obvious location for self-adaptive behavior is the cognitive operator. Due to the decoupling from the hard real-time processing complex processing steps for the self-optimization of a single OCM can be realized here. In a subsequent step they have to be enacted by influencing the behavior of the reflective operator and the controller accordingly. Due to the temporal decoupling the cognitive operator itself can remain outside the critical part of the software and it is sufficient to only consider all possible effects the cognitive operator may have on the reflective operator and controller. Usually, the reflective operator with all its configuration variants that can be steered by the cognitive operator is designed as a safety envelope. By ensuring that all configurations of the reflective operator work properly and abstracting from the possible configuration advices from the cognitive operator, we can still ensure safe self-optimizing behavior. However, what this scheme does not help to guarantee is that the self-optimization itself is successful. An OCM optimizing its reflective operator and controller by taking the long term changes of the controlled hardware due to abrasion into account is an example for such a self-optimization for a single OCM.

It is also possible to achieve self-optimization for a whole hierarchy of OCMs where the higher level reflective operators steer the subordinated OCMs to achieve a self-optimization for the whole hierarchy. Again the cognitive operators can play the role of driving the decisions. However, in this case the lower level OCMs and their cognitive operators are guided by the higher level OCMs which determine what are their optimization goals. Thus, here we got a complex interplay of local analysis and planning activities similar to a hierarchical optimization problem where the solutions identified at the higher level OCMs influence the search space that is considered at the lower level OCMs. Similar to the local case the reflective operators and their interaction can be studied without taking the complex behavior in the cognitive operators into account. The composition of the reflective operators become a safety envelop that protects the system against failures in the self-optimization. Again, the scheme does not help to guarantee that the self-optimization across a whole hierarchy of OCMs performs well and necessarily results in an improved system behavior. The energy management in a shuttle is an example for a self-optimization across a hierarchy of OCMs. While the overall OCM has to decide how much energy could be at most consumed by each lower level OCM to achieve the current higher level goals, the lower level OCMs try to optimize their energy consumption and the performance that can be achieved taking the constraints and precedences of the higher level OCMs into account only looking at their local scope.

Furthermore, also the agents and their peer-to-peer coordination can be employed to achieve a self-adaptation of the overall system. However, in this case two rather different cases can occur.

We can have the case that a behavior of a group of agents shows some emergent behavior due to the employed peer-to-peer protocols often referred to as self-organization. These emergent properties of the behavior may result in self-optimization but may also simply provide some required system properties. Here, the role interfaces provide some protection that can be exploited and depending on the complexity of the protocols guarantees for the emergent behavior are possible. The collision freedom for the shuttles later considered in this paper falls under this case.

In contrast to such emergent properties, the peer-to-peer coordination of the agents can also result in a self-optimization by exchanging information about the context such that the other agents benefit from this. Again, the role interfaces permit to ensure that the overall protocol works. However, as the data exchange and the related data processing can be rather complex, the scheme does not permit to guarantee that the information exchange effectively results in self-optimization. Furthermore, the scheme can not exclude that erroneous data result in unsafe behavior. Consequently, in this case no development-time solution is provided and problems with the exchanged data have to be detected at run-time and related fallback strategies must be available (c.f. runtime verification). Shuttles that exchange data about the track characteristics to improve their performance (c.f. [13]) are an example for this case of group-wise self-optimization. Note that for safety reasons besides the optimized controller that exploits the data about the track characteristics in addition a fallback controller that also works in case no data is available and a unit to detect whether the optimized controller does not perform well have been part of the related system design.

2.3 Modular and Compositional Verification

Our MDD approach takes the general model of Fig. 2 as an informal architectural basis. It provides a formal definition of arbitrary OCM hierarchies, their behavior as well as their peer-to-peer communication using a refined UML component model and a refined notion of statecharts including the definition of timing constraints and hybrid behavior. This definition is the input for our model checking approach, which uses standard real-time model checkers, but before using them decomposes the overall system in such a way that the individual parts can be checked separately. Additional checks that the interfaces are well-defined interfaces and that the components refine their interfaces then guarantee, that when composing the models the separately checked safety properties are also guaranteed for complex composed system.

As all safety and time-critical aspects are handled by the reflective operators and controllers, peer-to-peer communication (across the hierarchy) is also allowed between the different cognitive operators at different levels in the nested OCM hierarchies. This may facilitate complex planning and the required information exchange between different components, but the interface between a reflective and cognitive operator in each component and on each nesting level respectively will ensure that no unsafe behavior can result from the interaction with the cognitive operators.

2.4 Tool Support and Code Generation

To complete the approach, MUML is supported by the FUJABA Real-Time Tool Suite CASE tool [13–15] and includes a code-generation scheme [16–19] that maps all the high-level timing constraints of the models to underlying real-time operating system and scheduling technologies. Additional schedulability checks then ensure that the code executed on real-time operating systems provides the same safety guarantees as the models. Thus, the safety guarantees obtained via checking the models can also be transferred to the code level (c.f. [3]). In addition, an alternative mapping scheme onto Simulink and Stateflow Models [20] has been developed to facilitate also commercial simulation and code-generation tools.

3 Modeling

In this section we describe our solution for modeling OCM hierarchies as well as peer-to-peer networks as outlined in Fig. 2 using our extended UML component model.

3.1 The Hierarchical Component Model

To first capture OCM-like hierarchies, we describe a component model with a static structure adjusted to the needs of mechatronic systems. We then extend this component model to also cover the case of reconfiguration.

Component Structure. To support the coupling of time-continuous control behavior with discrete behavior, we extend the definition of ports in the UML component model. Ports may also be defined by time-continuous variables. While a signal is sent and received at discrete points in time (cf. SignalEvent in UML), a time-continuous variable has a well-defined value for each point in time.

As an example the MUML model of the OCM of the shuttle responsible for travelling either in convoy or stand alone mode is depicted in Fig. 3. The Shuttle component instance sh contains a AccelerationControl (AC) component instance ac representing the reflective operator and controller and a Planer component instance pl representing the cognitive operator. The reflective operator which mediates between the other two OCM components is represented by the shuttle component sh itself. This component computes the acceleration needed to achieve a specific goal (keeping a specified speed level or keeping a specified distance from the predecessor). The AccelerationControl component has five incoming continuous ports and one outgoing continuous port. We distinguish here between *permanent ports* and *optional ports*. The former are depicted by a black triangle and the latter by a white triangle to indicate that they are only active in some of the modes as introduced later when considering reconfiguration.

The incoming continuous ports are for the values current velocity v_{cur}, the current distance Δ_{cur}, and the velocity of the front shuttle v_{Front} provided by sensors, and the required velocity v_{req} and the required distance Δ_{req} which are parameterized reference inputs. The outgoing port sends acceleration values to the appropriate hardware actuator devices. In addition, the ac component contains discrete behavior to switch between keeping a certain distance and keeping the velocity at a constant level, and is thus a hybrid component.

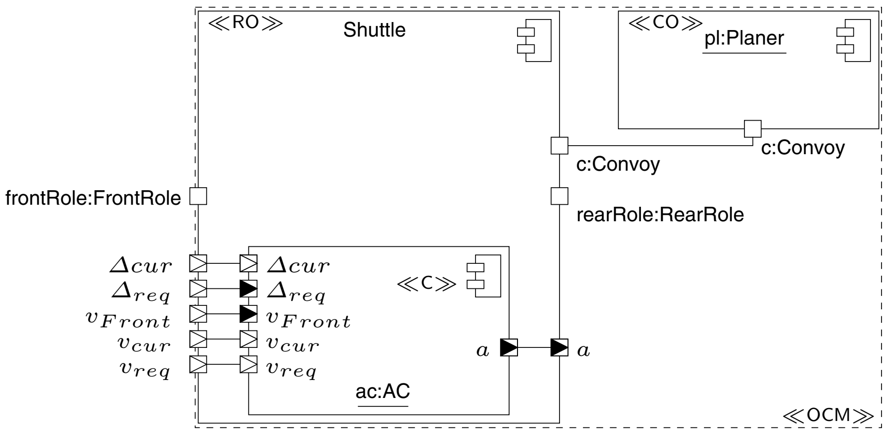


Fig. 3. Example for a component structure of a Shuttle OCM

The specification of component behavior is given by (extended) UML state machines called *Real-Time Statecharts* (RTSC) [16, 21, 22], which provide additional constructs to describe time-dependent behavior and information such as deadlines and worst case execution times (WCET). We introduce RTSC and their extension to a hybrid variant only informally here.

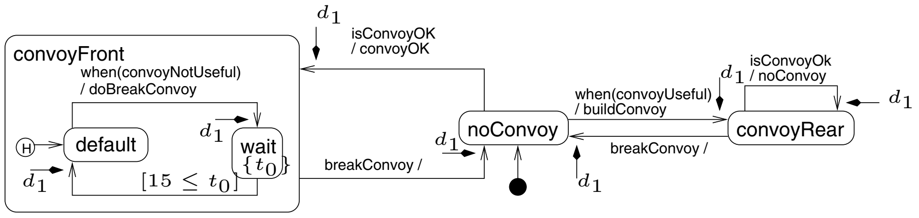


Fig. 4. Behavior of the Shuttle component

In Fig. 4 the internal behavior of the Shuttle component of Fig. 3 is defined by a RTSC. As an example for a typical real-time requirement a deadline interval d_1 is used to describe the state change from state noConvoy to state convoyFront which has to be finished within the given interval. Similarly, deadlines are defined to constrain the time an object may remain in a certain state. Transition guards may contain conditions which depend on the current value of a clock.

In general, clocks, time guards, and time invariants from timed automata [23, 24] are combined with expressive modeling concepts existing in UML state machines. RTSC are thus more expressive than plain or hierarchical timed automata models and permit emulation of limited UML state machine concepts for time such as after and when. In addition RTSC supports the definition of flexible timer conditions that must held over a series of states. Buffering of timing events is not needed and does not exist in this approach. This avoids non predictable effects which may exist in the UML state machines as well as their extensions which use external timers [25].

Extending RTSC to specify continuous behavior is done similarly to the basic hybrid automata approaches like [26–29] by the possibility of assigning a configuration of controllers to a particular state for *Hybrid RTSCs* (HRTSCs).

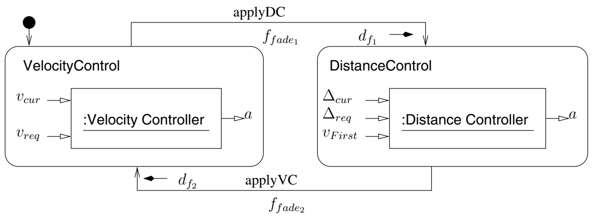


Fig. 5. Behavior of the AC component

An example of this is the hybrid behavior of an Acceleration Control (AC) component which is embedded into the Shuttle component. It consists of two discrete control modes which specify whether the shuttle is operating in velocity control mode or distance control mode respectively (see Fig. 5). Furthermore, it has continuous inputs and outputs. Depending on the active discrete mode, either the current and the required velocity are used as input, or the current and required distance to the front shuttle as well as the velocity of the first shuttle are used. The output a is the acceleration in both modes. In this example each configuration consists of one single feedback controller, while usually a configuration of subordinated blocks representing a number of (continuous) controllers might be assigned to each state.

Switching smoothly between different controllers requires the specification of an output *cross-fading function* (cf. [9]). In our example we have the fading functions f_{fade_1} and f_{fade_2} and a minimal and a maximal *fading duration* (d_{f_1} respectively d_{f_2}), which specify how the outputs of the two controllers have to be faded when changing the controller.

In the example depicted in Fig. 5, the state-dependent continuous behavior is specified by blocks assigned to the states VelocityControl and DistanceControl. If an RTSC contains hierarchies and thus comprises state configurations rather than single states, the assigned controller configuration is the union of all configurations assigned to the states of the current state configuration.

Component Structure with Reconfiguration. While the outlined static component model supports the specification of nested component structures, it does not cover the possibility of components having changing input/output interfaces depending on the current system state. As an example take the AC component of Fig. 5. If you consider the shuttle component, then the AC component should be in state VelocityControl only if the shuttle is in state noConvoy or convoyFront. In this case the AC component requires two inputs. If the shuttle is in state convoyRear, however, the AC component should be in state DistanceControl and requires three inputs.

To cover nested components with changing input/output (i.e. OCM hierarchies with reconfiguration), we introduce an extension of the known concept of hybrid behavior that assigns a configuration of embedded (possibly hybrid) component instances to each state instead of control behavior only. The related HRTSC depicted in Fig. 6 extends Fig. 4 accordingly.

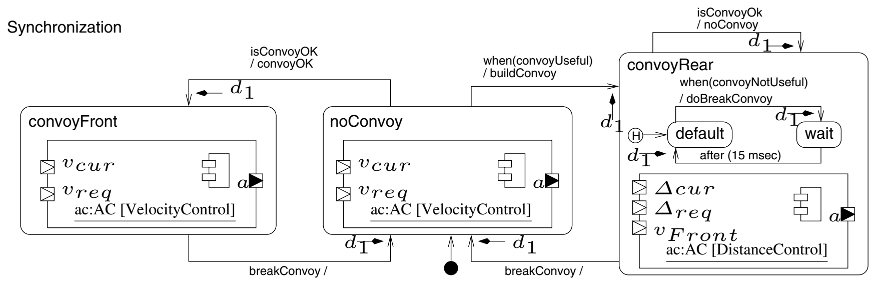


Fig. 6. Behavioral embedding

Fig. 6 depicts the orthogonal Synchronization state, whose sub-states embed different configurations, each consisting of one AC instance ac and its current mode and continuous interface. It is thus specified that ac has to be in mode DistanceControl when Synchronization is in state convoyRear. If Synchronization is in state noConvoy or convoyFront, ac has to be in mode VelocityControl. Consequently a state change within the orthogonal Synchronization state implies a mode change in its embedded ac component.

Referring to the example in Fig. 6, the internals of a component behavior, such as the AC component, need not be known in order to embed a component behavior specification into the HRTSC of its superior component, i.e. assign it to certain states. Internal in this case are the definitions of the controllers as given in Fig. 5. Rather, it is enough to specify an interface statechart for each component that defines the externally relevant behavior. Fig. 7 gives an example of the interface statechart of the AC component. Note that continuous ports only available on a subset of the modes become *optional ports* while those ports supported become *permanent ports* (cf. also Fig. 3).

The externally relevant behavior includes the definition of the different control modes, the modes' continuous inputs and outputs as well as the dependencies between outputs and inputs, and the deadline information for switches between the control modes. The specific control strategy employed in each mode, whether fading is required for a transition, the kind of fading function applied for a

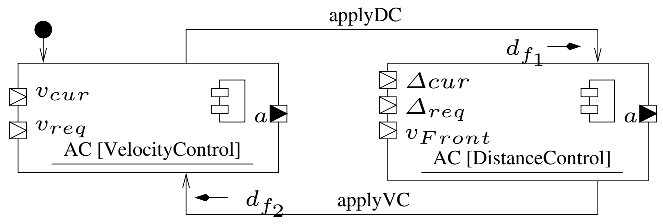


Fig. 7. Interface statechart of the AC component

transition, and which embedded components are active in each mode are implementation details not relevant to the external view of an interface statechart.

In each component may exist states that are related to potentially unsafe situations. Therefore, we refer to a component as locally safe if such states can be excluded for a given context. Also when the components do not work properly together or when the reconfiguration across multiple levels via the interface statecharts does result in any violations of the specified timing constraints, the assumptions of the composed components are not fulfilled and thus their local safety is no longer guaranteed. In the later considered models, in case of such an incompatibility the composed behavior will exhibit a deadlock.[1] Thus we can conclude that a hierarchical system of components is *safe* as long as the components are locally safe and the composition does not result in a deadlock.[2]

In our example the Shuttle builds a hierarchy of components, where each Shuttle instance contains a single supervised embedded component instance of type AC. The local safety guarantees that each local OCM is safe. The deadlock freedom guarantees that the complex reconfiguration fulfills all timing constraints.

Definition 1. *The overall safety of a hierarchical system is given if all component are locally safe and the overall behavior does not contain any deadlock.*

In our example we have the Shuttle agents with a single supervised embedded component instance of type AC. Within the shuttles the switching has to adhere to the timing constraints for cross-fading the outputs and the commitments concerning braking in different collaborations must be not in conflict. The deadlock freedom guarantees that the complex reconfiguration fulfills all timing constraints for the reconfiguration present at the different levels of the hierarchy.

The remaining part of the architecture, consisting of the cognitive operators and their interconnections, is additionally covered by related components that

[1] This concerns the usual definition of a deadlock but also so-called time stopping deadlocks. A time stopping deadlock means that a system cannot progress due to an inconsistency in the definition of timing constraints of transitions or states.

[2] We assume here that required non-local safety properties that relate to a number atomic components in a system are considered part of a local safety properties of a hierarchical element that contains all in the required safety property involved elements.

are connected with the safety-critical hierarchical core only via unsafe ports that decouple the core from the rest.

3.2 The Peer-to-Peer Coordination Model

Besides the hierarchical component structures and their hybrid behavior as addressed in the last section, MUML specifications can also capture peer-to-peer interaction of autonomous mechatronic systems (cf. Fig. 2). At the peer-to-peer level the interaction between components is specified in MUML by so-called coordination patterns. At this level no hybrid behavior exists anymore because communication between components is only based on discrete events and corresponding actions. Therefore these patterns can be described by a refinement of the loosely defined collaboration and pattern concepts in UML using RTSC to specify the behavior of roles and connectors.

Real-Time Coordination Pattern. Real-time coordination patterns allow to specify the interaction between multiple mechatronic agents using UML collaborations so that the behavior is rigorously defined. Therefore a real-time coordination pattern includes a description of the roles involved agents may play . The agents can interact only via these roles and connectors that connects them. Each role and connector in turn are specified by an RTSC that captures the behavior permitted and expected from each role as well as the communication medium.

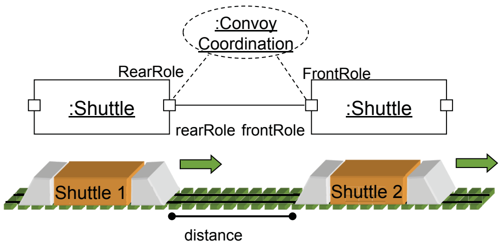


Fig. 8. Component Instance Diagram and Pattern Instance

The communication between two shuttles necessary to build a convoy is one such real-time coordination pattern. Fig. 8 shows a ConvoyCoordination pattern instance between two shuttles. It defines a drastically simplified protocol for building and breaking convoys based on two roles, namely the rear role and the front role (see Fig. 9).

Initially, both roles are in state noConvoy::default, which means that they specify the situation where a shuttle is not a member of a convoy. The rear role non-deterministically chooses whether to propose building a convoy or not. After choosing to propose a convoy, a message is sent to another shuttle, or rather its front role instantiation. The front role non-deterministically chooses to reject or to accept the proposal after at most 1000 msec. In the first case, both statecharts

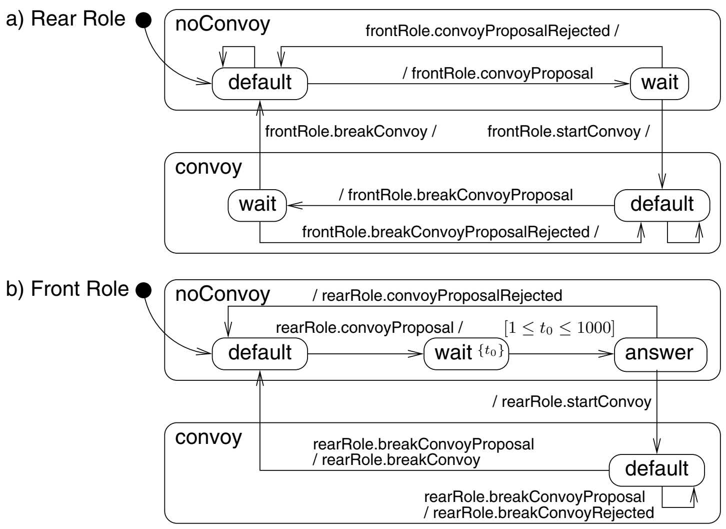

a) Rear Role
noConvoy
frontRole.convoyProposalRejected /
default
/ frontRole.convoyProposal
wait
frontRole.breakConvoy /
frontRole.startConvoy /
convoy
wait
/ frontRole.breakConvoyProposal
default
frontRole.breakConvoyProposalRejected /
b) Front Role
noConvoy
/ rearRole.convoyProposalRejected
rearRole.convoyProposal /
[1 ≤ t0 ≤ 1000]
default
wait {t0}
answer
/ rearRole.startConvoy
convoy
rearRole.breakConvoyProposal
/ rearRole.breakConvoy
default
rearRole.breakConvoyProposal
/ rearRole.breakConvoyRejected

Fig. 9. RTSC of the RearRole role and the FrontRole role

revert to the noConvoy::default state. In the second case, both roles switch to the convoy::default state.

Eventually, the rear shuttle non-deterministically chooses to propose breaking the convoy and sends this proposal to the front shuttle. The front shuttle non-deterministically chooses to reject or accept that proposal. In the first case, both shuttles remain in convoy-mode. In the second case, the front shuttle replies with an approval message and both roles switch into their respective noConvoy::default states.

The connector which represents the wireless network does not need to be specified by an explicit statechart specification here, but instead by its QoS characteristics such as throughput, maximal delay etc. in the form of connector attributes. In our example we assume that the connector forwards incoming signals with a delay of between 1 to 5 msec. The connector is unsafe in the sense that it might fail at any time so that we set our specific QoS characteristic reliable to false.

The specification of safety properties is given by declarative constraints which are defined using temporal logic using a state-based temporal extension of the Object Constraint Language (OCL) called RT-OCL [30]. As the examples in this paper only contain formulas in pure OCL, we further omit any details of RT-OCL here.

A safety property of this pattern is that a shuttle should only make an emergency brake when it is not taking the front position in a convoy. Using an atomic proposition CanBrakeFully, which specifies whether a shuttle can brake with full strength, the required safety property is that when an implementation of the rear role is in state convoy the atomic proposition CanBrakeFully must be true and when an implementation of the front role is in state convoy the atomic proposition CanBrakeFully must be false. The following OCL role invariants ψ_1 and ψ_2 are used to describe these restrictions.[3]

```
context <comp> inv: <frontRole>.oclInState(convoy) implies not self.CanBrakeFully    (1)
```

```
context <comp> inv: <rearRole>.oclInState(convoy) implies self.CanBrakeFully    (2)
```

Atomic Agent. When defining the behavior of an agent like a shuttle using predefined patterns such as e.g. the ConvoyCoordination pattern mentioned above, the predefined role behavior has to be refined and synchronized. The following example illustrates this step.

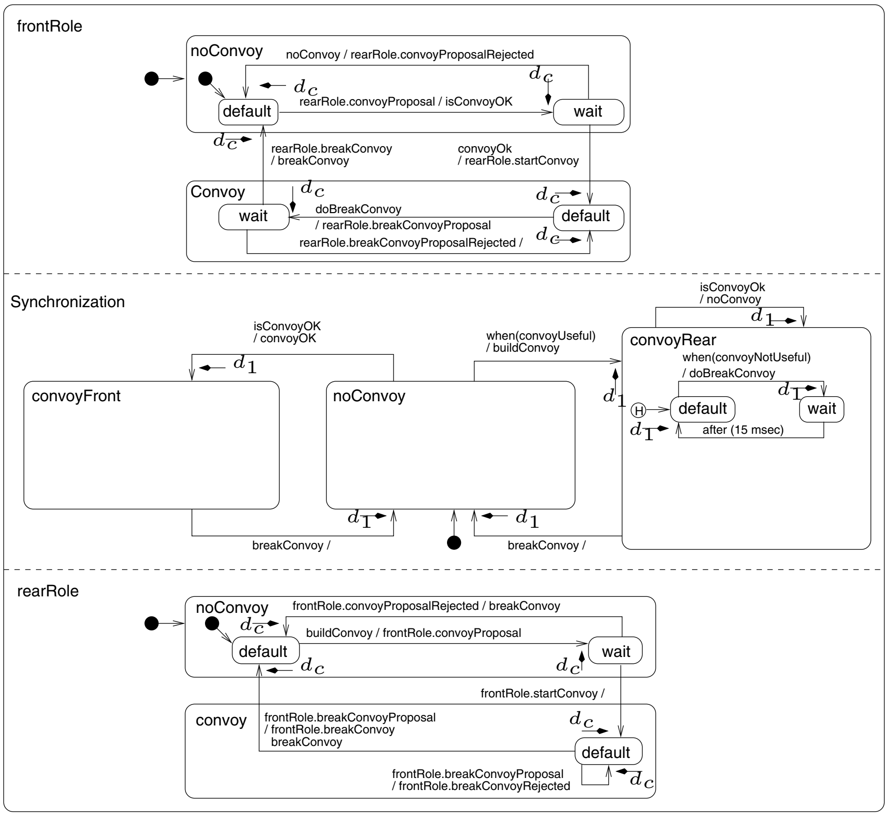

frontRole
noConvoy
noConvoy / rearRole.convoyProposalRejected
default
rearRole.convoyProposal / isConvoyOK
wait
rearRole.breakConvoy / breakConvoy
convoyOk / rearRole.startConvoy
Convoy
wait
doBreakConvoy / rearRole.breakConvoyProposal
default
rearRole.breakConvoyProposalRejected /
Synchronization
isConvoyOK / convoyOK
convoyFront
noConvoy
when(convoyUseful) / buildConvoy
isConvoyOk / noConvoy
convoyRear
when(convoyNotUseful) / doBreakConvoy
H
default
wait
after (15 msec)
breakConvoy /
breakConvoy /
rearRole
noConvoy
frontRole.convoyProposalRejected / breakConvoy
buildConvoy / frontRole.convoyProposal
default
wait
frontRole.startConvoy /
convoy
frontRole.breakConvoyProposal / frontRole.breakConvoy breakConvoy
default
frontRole.breakConvoyProposal / frontRole.breakConvoyRejected
d_C
d_1

Fig. 10. Behavior of the Shuttle agent

[3] The context <comp> enclosed in angle brackets is employed here as a placeholder for the component which realizes the role via one of its ports.

Fig. 10 depicts the behavior of the Shuttle agent from Fig. 3. The HRTSC consists of three orthogonal states FrontRole, RearRole and Synchronization.

FrontRole and RearRole describe the port behavior. They are refinements of the role behaviors in Fig. 9 and specify in detail the communication that is required to build and to break convoys. Syntactical refinement rules or a special checking procedure, outlined later in Section 4, are used to ensure that the refinement does not invalidate any safety properties which have been verified for the (non-refined) pattern already. Basically, only the non-determinism possibly still existing in a RTSC defining a role is reduced by the refinement.

An additional internal HRTSC is used to specify the synchronization. In our example, Synchronization coordinates the communication and is responsible for initiating and breaking convoys. The three sub-states of Synchronization model whether the shuttle is in the convoy at the first position (convoyFront), at second position (convoyRear) or whether no convoy is built at all (noConvoy).

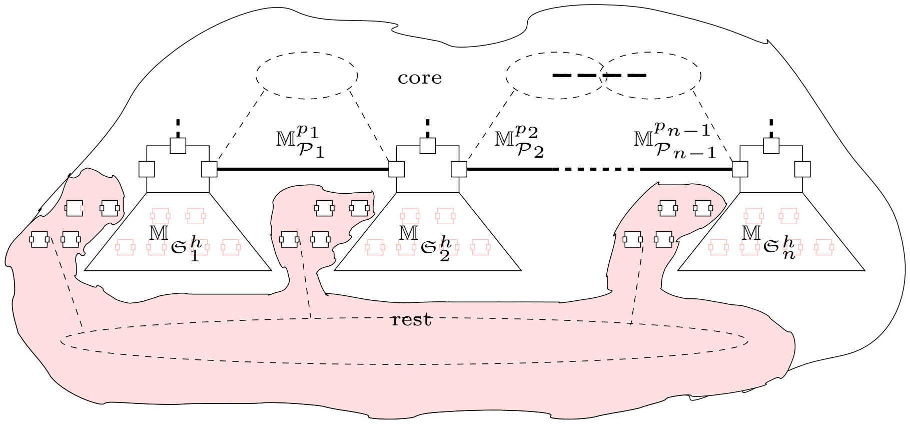

core
$\mathbb{M}^{p_1}_{\mathcal{P}_1}$
$\mathbb{M}^{p_2}_{\mathcal{P}_2}$
$\mathbb{M}^{p_{n-1}}_{\mathcal{P}_{n-1}}$
$\mathbb{M}_{\mathfrak{S}^h_1}$
$\mathbb{M}_{\mathfrak{S}^h_2}$
$\mathbb{M}_{\mathfrak{S}^h_n}$
rest

Fig. 11. Peer-to-peer composition of agents (upper part), hierarchies within the agents (middle part) and decomposition into a safety-critical core and a arbitrarily structured rest (bottom and middle part)

A system built by a set of atomic agents and pattern instances (Fig. 11 upper part) then describes the free peer-to-peer interaction of the agents by pattern instances.

Definition 2. *A peer-to-peer system is* safe *if for the behavior ensures that*

- *all agents/components are locally safe,*
- *no deadlock can occur,*
- *all RT-OCL constraints of pattern instances are fulfilled, and*
- *all OCL role invariants of agent instances are fulfilled.*

In our example we have the Shuttle agents connected via instances of the ConvoyCoordination pattern. The RT-OCL constraints guarantee that no collision is possible for shuttles connected by ConvoyCoordination pattern instances. The OCL role invariants ensure that the agents behave consistently with the guarantees related to their roles, e.g. a shuttle will brake accordingly when the state of the role of the ConvoyCoordination pattern requires it. However, the pure peer-to-peer system does not cover the embedding of subordinated components such as the AC component.

Hierarchical Agents with Reflective Operator. In case of a hierarchical agent, besides refining and synchronizing the assigned role behavior, the reconfiguration of the embedded hybrid component hierarchy also has to be specified.

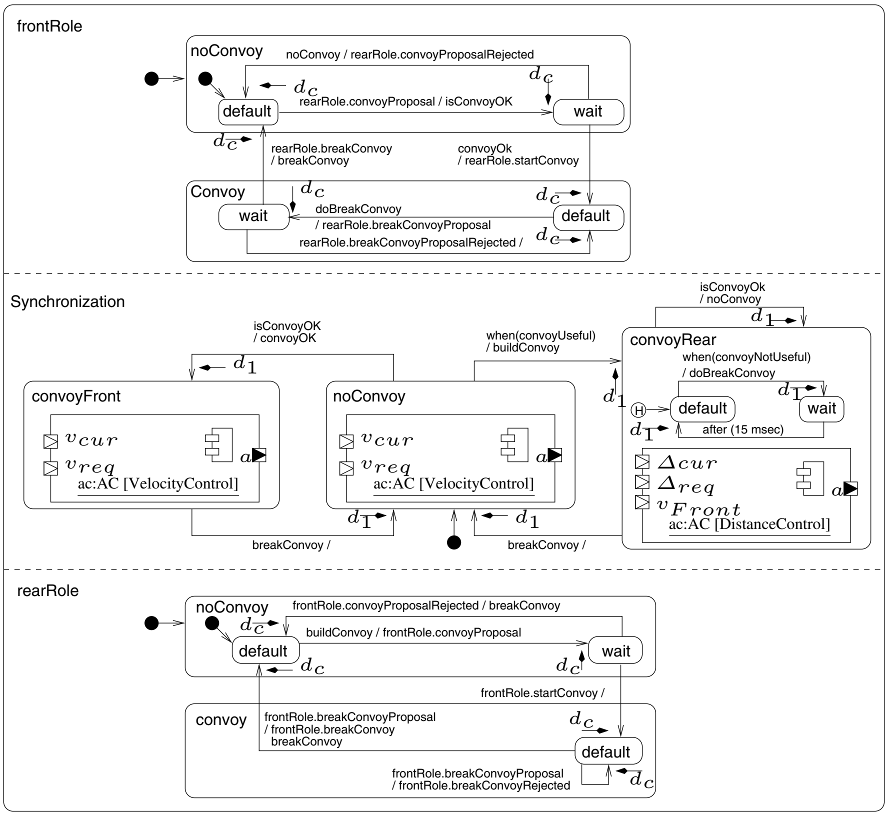

frontRole
noConvoy
noConvoy / rearRole.convoyProposalRejected
rearRole.convoyProposal / isConvoyOK
default
wait
rearRole.breakConvoy / breakConvoy
convoyOk / rearRole.startConvoy
Convoy
wait
default
doBreakConvoy / rearRole.breakConvoyProposal
rearRole.breakConvoyProposalRejected /
Synchronization
isConvoyOK / convoyOK
when(convoyUseful) / buildConvoy
isConvoyOk / noConvoy
convoyFront
noConvoy
convoyRear
when(convoyNotUseful) / doBreakConvoy
default
wait
after (15 msec)
ac:AC [VelocityControl]
ac:AC [VelocityControl]
ac:AC [DistanceControl]
breakConvoy /
breakConvoy /
rearRole
noConvoy
frontRole.convoyProposalRejected / breakConvoy
buildConvoy / frontRole.convoyProposal
default
wait
frontRole.startConvoy /
convoy
frontRole.breakConvoyProposal / frontRole.breakConvoy breakConvoy
default
frontRole.breakConvoyProposal / frontRole.breakConvoyRejected

Fig. 12. Behavior of the Shuttle agent

An instance of the hybrid component type AC is assigned to the different three sub-states of Synchronization. In state convoyFront and state noConvoy the embedded controller is run in mode VelocityControl, but in state convoyRear mode DistanceControl is used to ensure a proper distance to the other shuttle (Fig. 12).

Our approach for the modeling of a hierarchy of OCMs, as depicted in Fig. 2 without cognitive operators, is based on the observation that it can be modeled as hierarchal agents that coordinate themselves by pattern instances. While the free peer-to-peer interaction of the top-level OCMs can be captured by pattern instances, a hierarchical system of configurable components can be used to cover the hierarchies of reflective operators and controllers (Fig. 11 middle part).

Hierarchical Agents with Cognitive Operator. However, the cognitive operators do not really fit into this picture as they do not fit into the hard real-time processing scheme established by the hierarchy of reflective operators but form their own, often less strict, structures. For the cognitive operators, verification is usually not feasible, or at least extremely expensive, due to their intelligent behavior. Their tight integration within the safety-critical and hard real-time part of the system is thus problematic. However, we can require the boundary ports to be *unsafe ports* to express that you cannot rely on the offered interaction. Therefore, such unsafe ports can be used to decouple the safety-critical parts of the system from those for which safe operation cannot be guaranteed such as the cognitive operators.

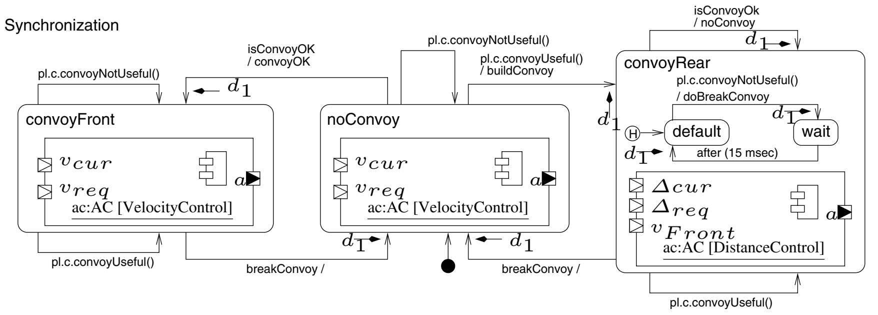


Fig. 13. Decoupling of the unreliable, soft real-time, planning component

In Fig. 13, the unsafe port pl.c connecting the soft real-time planning of the cognitive operator with the safety-critical, hard real-time processing of the reflective operator as depicted in Fig. 3) is used to steer proposing convoys. If the planning component suggests building a convoy with the rear shuttle, it indicates this by sending a pl.c.convoyUseful() message. If it deduces that a convoy with the rear shuttle should be broken up it sends pl.c.convoyNotUseful(). The depicted behavior is therefore able to handle both messages in any state so that erroneously sent messages of the planning component cannot result in unsafe shuttle behavior.

Our approach for the modeling of the safety-critical core of a hierarchy of OCMs as depicted in Fig. 2 employs hierarchical agents and pattern instances as outlined in Fig. 11. The remaining part of the architecture consisting of the cognitive operators and their interconnections is additionally covered by related components that are connected with the safety-critical core only via unsafe ports as depicted in Fig. 11.

Definition 3. *A system with included core system is* safe *if the core behavior ensure that*

- *all agents/components are locally safe,*
- *no deadlock can occur,*
- *all RT-OCL constraints of pattern instances of the core are fulfilled, and*
- *all OCL role invariants of agent instances of the core are fulfilled.*

In our example we have the Shuttle agents connected via instances of the ConvoyCoordination pattern. For each Shuttle instance, a single supervised embedded component instance of type AC exists. The deadlock freedom guarantees that the complex reconfiguration fulfills all timing constraints. The RT-OCL constraints guarantee that no collision is possible for shuttles connected by ConvoyCoordination pattern instances. The OCL role invariants ensure that the agents behave consistently with the guarantees related to their roles, e.g. a shuttle will brake accordingly when the state of the role of the ConvoyCoordination pattern requires this.

4 Modular and Compositional Verification

This section outlines how the modular and compositional formal verification of self-optimizing systems developed with the MUML approach can guarantee safety. These results are based on the rigorous definitions for the employed concepts also provided in this section (please note that some more fundamental definitions and additionally required consistency and well-formedness conditions can be found in [3]).

The key aspect of our approach is that our notion of a consistent core system enables a modular and compositional verification where only the single component types and patterns with their interfaces resp. embeddings are considered. For hierarchal systems (including agent subsystems) we can exploit the modular structure to derive the safety of all included component instances only looking at the single component types as well as their interface and embeddings (see Section 4.1 and Theorem 1). A compositional scheme also allows the safety of all pattern instances to be ensured only by looking into the patterns and their roles and connectors as well as the conformance of all components with respect to the ports which are attached to the roles (see Section 4.2 and Theorem 2). Due to the manner in which the core is decoupled from the rest of the system via unsafe ports, we can further show that this result cannot be invalidated by the rest of the system (see Section 4.3 and Theorem 3). These separate results can be employed to guarantee that a complete system is safe via modular and compositional verification (see Section 4.3 and Corollary 1), essentially only by looking at the types and without considering the complete system or any larger subsystem with all its component and pattern instances explicitly. Consequently, existing model checking techniques can be employed as the usual state explosion due to parallel composition of multiple instances is avoided.

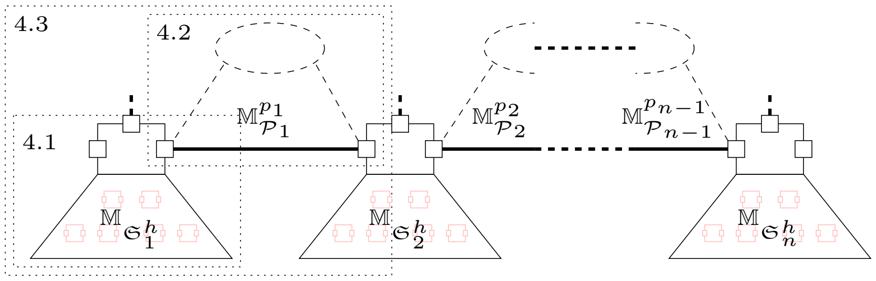

4.3
4.2
4.1
$\mathbb{M}^{p1}_{\mathcal{P}1}$
$\mathbb{M}^{p2}_{\mathcal{P}2}$
$\mathbb{M}^{p_{n-1}}_{\mathcal{P}_{n-1}}$
$\mathbb{M}_{\mathfrak{S}^h_1}$
$\mathbb{M}_{\mathfrak{S}^h_2}$
$\mathbb{M}_{\mathfrak{S}^h_n}$

Fig. 14. Decomposition of the core system for the verification

4.1 Hierarchical Component Model

Syntax, Semantics and Safety. In the following we describe the resulting behavior using automata ($\mathbb{M}$) as well as their parallel composition ($\|$). The formal semantics are defined in [3]. We also refer to [3] for the additional required consistency and well-formedness conditions between the component model, structure, and behavior.

An OCL or RT-OCL property ϕ is well-defined for a behavior $\mathbb{M}$ if ϕ only refers to properties of the states of $\mathbb{M}$. In the following, we describe that an OCL or RT-OCL property ϕ holds for a given real-time behavior $\mathbb{M}$ by $\mathbb{M} \models \phi$. In addition, the special symbol δ is used to specify that a deadlock exists. We further restrict the considered RT-OCL properties to compositional ones that were preserved by the parallel composition ($\|$) if the composition result is deadlock free (c.f. [6,7]).

Basis for the hierarchical component model are *reconfigurable component types* defined as follows:

Definition 4. *A* reconfigurable component type $\mathcal{C} = (\mathcal{S}_\mathcal{C}, \mathbb{M}_\mathcal{C}, \phi_\mathcal{C})$ *is given by a mode-dependent internal structure* $\mathcal{S}_\mathcal{C}$ *for a mode set* L*, an internal behavior* $\mathbb{M}_\mathcal{C}$ *with mode set* L*, and a local safety property* $\phi_\mathcal{C}$*.*

A *mode-dependent internal structure* for a given mode set L is a tuple $\mathcal{E} = (\mathcal{I}_\mathcal{S}, \mathcal{E}_\mathcal{S}, map_\mathcal{S})$, where $\mathcal{E}_\mathcal{S}$ is a function describing the *mode-dependent embedded components* for a given mode set L (it assigns to each state $l \in L$ a set of embeddings that are pairs of the form $(o, (\mathbb{M}, l))$ where o is an occurrence name and $(\mathbb{M}, l)$ is a pair consisting of an interface statechart $\mathbb{M}$ and one of its modes l). $\mathcal{I}_\mathcal{S}$ is a function describing the *mode-dependent interface* for a given mode set L (it assigns an interface consisting of a set of pairs of unique port names and port declarations to each state $l \in L$), and $map_\mathcal{S}$ is a *mode-dependent mapping* for a given mode set L (it assigns a mapping describing the connectors between ports in each mode to each state $l \in L$). More details can be found in [3].

The behavior $\mathbb{M}_\mathcal{C}$ also includes the forwarding behavior related to each mode in L which ensures that signals from the embedded components are routed as specified in the HRTSC (see [3]).

A reconfigurable hierarchical subsystem consisting of a number of reconfigurable components as depicted in Fig. 11 (middle) is then constructed as follows:

Definition 5. *A* hierarchical subsystem with reconfiguration $\mathfrak{S}^h$ *is a tuple* $(O_{\mathfrak{S}^h}, c_{\mathfrak{S}^h})$ *with* $O_{\mathfrak{S}^h} \subseteq \wp(\mathcal{N}_{\mathcal{C}}^+)$ *a set of instance names and* $c_{\mathfrak{S}^h}$ *a function which maps each instance* $o \in O_{\mathfrak{S}^h}$ *to a related reconfigurable component type.*

The *behavior* and *safety property* of a such hierarchical subsystem with reconfiguration $\mathfrak{S}^h = (O_{\mathfrak{S}^h}, c_{\mathfrak{S}^h})$ is then given by

$$\mathbb{M}_{\mathfrak{S}^h} := \mathop{\|}_{o \in O_{\mathfrak{S}^h}} \mathbb{M}^o_{c_{\mathfrak{S}^h}(o)} \qquad \phi_{\mathfrak{S}^h} := \bigwedge_{o \in O_{\mathfrak{S}^h}} \phi^o_{c_{\mathfrak{S}^h}(o)}.$$

For such a hierarchical subsystem with reconfiguration we have to ensure that all components are locally safe and have to exclude that the interaction or timing constraints invalidates the local safety of the components. Therefore, we define it as safe if its behavior guarantees the local safety of the components and is deadlock free (by providing a formal version for Definition 1).

Definition 6. *A hierarchical subsystem with reconfiguration* $\mathfrak{S}^h$ *is* safe *if its behavior* $\mathbb{M}_{\mathfrak{S}^h}$ *is well-formed, ensures local safety (*$\mathbb{M}_{\mathfrak{S}^h} \models \phi_{\mathfrak{S}^h}$*), and deadlock free (*$\mathbb{M}_{\mathfrak{S}^h} \models \neg\delta$*).*

Modular Verification. We exploit the well-defined hierarchy of an agent to prove local safety and deadlock freedom. Thus, we first look at the atomic components and the bottom of the hierarchies, then the embedding steps, and finally demonstrate that for the required behavioral consistency of the whole hierarchy, these two local checks are sufficient. To denote the behavior that results when restricting a hybrid reconfiguration automata to the real-time behavior and clocks we use an operator $RT()$ (see [3]).

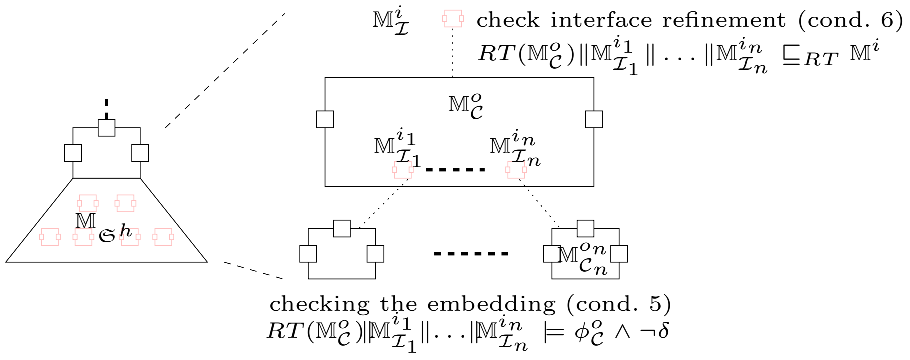
Atomic Component Types. The locally safe operation of a component type $\mathcal{C}$ requires that component behavior $\mathbb{M}_{\mathcal{C}}$ itself cannot result in a deadlock. A component without embedding is therefore *locally safe* if

$$\mathbb{M}_{\mathcal{C}} \text{ well-formed} \quad \wedge \quad RT(\mathbb{M}_{\mathcal{C}}) \models \phi_{\mathcal{C}} \wedge \neg\delta. \tag{3}$$

To further ensure that, in a given system, all embedded occurrences represented by their interface statecharts $\mathbb{M}_I^i$ are *behaviorally consistent* with the related embedded components with respect to behavior $\mathbb{M}_{\mathcal{C}}^o$, we have to ensure that real-time refinement holds:

$$RT(\mathbb{M}_{\mathcal{C}}) \sqsubseteq_{RT} \mathbb{M}_I^i. \tag{4}$$

Model checking is employed to fully automate the checking of condition 3. In many cases the required refinement condition 4 means the safe transfer of timing



$\mathbb{M}^i_{\mathcal{I}}$
check interface refinement (cond. 6)
$RT(\mathbb{M}^o_{\mathcal{C}})\|\mathbb{M}^{i1}_{\mathcal{I}_1}\|\ldots\|\mathbb{M}^{in}_{\mathcal{I}_n} \sqsubseteq_{RT} \mathbb{M}^i$
$\mathbb{M}^o_{\mathcal{C}}$
$\mathbb{M}^{i1}_{\mathcal{I}_1}$
$\mathbb{M}^{in}_{\mathcal{I}_n}$
$\mathbb{M}_{\mathfrak{S}}h$
$\mathbb{M}^{on}_{\mathcal{C}_n}$
checking the embedding (cond. 5)
$RT(\mathbb{M}^o_{\mathcal{C}})\|\mathbb{M}^{i1}_{\mathcal{I}_1}\|\ldots\|\mathbb{M}^{in}_{\mathcal{I}_n} \models \phi^o_{\mathcal{C}} \wedge \neg\delta$

Fig. 15. Hierarchical verification via interface abstraction and component-wise checks

constraints from one level to the next one, and can be guaranteed following syntactical refinement rules (cf. [9]).

In more complex cases, model checking also has to be employed (cf. [31–33]).[4]

Hierarchical Component Types. In a strict hierarchical system the continuous model for a state of the system can become *undefined* if the resulting continuous equations contain a cycle. Refinement guarantees that any dependency between an input and output in the behavior of embedded component occurrences $\mathbb{M}^{o_1}_{\mathcal{C}_1}, \ldots, \mathbb{M}^{o_n}_{\mathcal{C}_n}$ of a component is also present in their interface statecharts $\mathbb{M}^{i_1}_{I_1}, \ldots, \mathbb{M}^{i_n}_{I_n}$. Therefore, checking that the interface statechart combined with the embedding HRTSC $\mathbb{M}^o_{\mathcal{C}}$ is well-formed and sufficient to exclude cycles in the resulting continuous models.

Additionally, the locally safe synchronization of the fading-durations in the different components has to be ensured. We need to therefore ensure that the composition of the component behavior with the embedded interface statecharts cannot result in a deadlock.

A component with embedding is therefore only *locally safe* for the accordingly relabeled embedded interface statecharts $\mathbb{M}^{i_1}_{I_1}, \ldots, \mathbb{M}^{i_n}_{I_n}$ if

$$\mathbb{M}^o_{\mathcal{C}}\|\mathbb{M}^{i_1}_{I_1}\|\ldots\|\mathbb{M}^{i_n}_{I_n} \text{ well-formed} \wedge RT(\mathbb{M}^o_{\mathcal{C}})\|\mathbb{M}^{i_1}_{I_1}\|\ldots\|\mathbb{M}^{i_n}_{I_n} \models \phi^o_{\mathcal{C}} \wedge \neg\delta. \qquad (5)$$

As with the atomic case in condition 4 we also have to show that the interface statechart $\mathbb{M}^i_I$ alone is a real-time abstraction of the HRTSC $\mathbb{M}^o_{\mathcal{C}}$ combined with the interface statecharts of all subcomponents. For *behavioral consistency* it is necessary that the real-time abstraction of the component behavior in form of the interface statechart $\mathbb{M}^i_I$ in fact refines the behavior which results when we

[4] As with the general form of hybrid systems considered here reachability is undecidable [34] and we cannot expect to find an automatic solution for the general problem. However, the developed techniques cover all relevant cases in practice for MUML.

compose the component behavior $\mathbb{M}^o_{\mathcal{C}}$ with all accordingly relabeled embedded interface statecharts $\mathbb{M}^{i_1}_{I_1}, \dots, \mathbb{M}^{i_n}_{I_n}$:

$$RT(\mathbb{M}_{\mathcal{C}}) \| \mathbb{M}^{i_1}_{I_1} \| \dots \| \mathbb{M}^{i_n}_{I_n} \sqsubseteq_{RT} \mathbb{M}^{i}_{I}. \tag{6}$$

Model checking is employed to fully automate the checking of condition 5 like in case of checking condition 3. Therefore, the interface statecharts are considered in addition to the component behavior.

In case of simple interface statecharts condition 6 can be checked at the syntactical level. It only has to be considered whether each transition in the HRTSC and the related transitions in the interface statecharts of the aggregated subcomponents are consistent (cf. [9]).

Assume the example in Fig. 12 and 6 which specifies that a change from state noConvoy to convoyRear has to be finished after 200 msec and that this change implies a change of the embedded AC component from VelocityControl to DistanceControl. Then, in Fig. 7, the minimal fading duration may not be above 200 msec.

Besides this purely syntactical check for simple interface statecharts, the embedding of more general notions of interface statecharts can be addressed using model checking (cf. [31–33]).

Thus, for components with embedded components, either syntactical checks or more advanced model checking techniques can be employed to check condition 6.

Hierarchical Systems. Checking local safety for all component types embedded in one hierarchal component guarantees that the whole hierarchy cannot become deadlocked. The following theorem proves that the local safety and behavioral consistency, which has been checked for each embedding, is sufficient to ensured that the behavior, which results when the component and all its direct subcomponents $\mathbb{M}^{o_1}_{\mathcal{C}_1}, \dots, \mathbb{M}^{o_n}_{\mathcal{C}_n}$ are considered and is a refinement of the HRTSC $\mathbb{M}^o_{\mathcal{C}}$ ($\sqsubseteq_{RT}$).

Theorem 1. *For a consistent hierarchical subsystem $\mathfrak{S}^h = (O^c_{\mathfrak{S}^h}, c_{\mathfrak{S}^h})$ with unique top-level component $o \in O_c \cap \mathcal{N}_{\mathcal{C}}$, only locally safe embedded components $o_1, \dots, o_n$ ($O_c = \{o, o_1, \dots, o_n\}$; see condition 3 and 5) and where all embeddings are behaviorally consistent (see condition 4 or 6) holds for the real-time abstraction $RT(\mathbb{M}^o_{c_{\mathfrak{S}^h}(o)})$ of the HRTSC $\mathbb{M}^o_{c_{\mathfrak{S}^h}(o)}$:*

$$RT(\mathbb{M}^o_{c_{\mathfrak{S}^h}(o)} \| \mathbb{M}^{o_1}_{c_{\mathfrak{S}^h}(o_1)} \| \dots \| \mathbb{M}^{o_n}_{c_{\mathfrak{S}^h}(o_n)}) \sqsubseteq_{RT} RT(\mathbb{M}^o_{c_{\mathfrak{S}^h}(o)}) \text{ and} \tag{7}$$

$$RT(\mathbb{M}^o_{c_{\mathfrak{S}^h}(o)} \| \mathbb{M}^{o_1}_{c_{\mathfrak{S}^h}(o_1)} \| \dots \| \mathbb{M}^{o_n}_{c_{\mathfrak{S}^h}(o_n)}) \models \phi^o_{c_{\mathfrak{S}^h}(o)} \wedge \neg\delta \tag{8}$$

Proof. (sketch) We can show the required result by induction over the depth over the hierarchical component structure using condition 4 and condition 6, substituting the component step-wise for the component interfaces for the whole subsystem beneath. For each such substitution we can conclude from the condition for the type that the same relation holds for all specific instances. Thus, the top-level component is refined by the whole hierarchical component concerning its *real-time behavior, based on the fact that $\sqsubseteq_{RT}$ is a precongruence for $\|$, and that we only require a finite number of substitution steps. Furthermore, $\phi^o_{c_{\mathfrak{S}^h}(o)} \wedge \neg\delta$ is guaranteed as the safety properties in $\phi^o_{c_{\mathfrak{S}^h}(o)}$ are compositional and $\neg\delta$ is guaranteed as deadlock freedom has been checked for the top component.*

4.2 Peer-to-Peer Coordination Model

Syntax, Semantics and Safety. The main ingredients of peer-to-peer systems are the real-time coordination patterns and agents. We will also cover the integration of the pure peer-to-peer scheme with the hierarchies present in the agents as well as the decoupled cognitive operators via unsafe ports.

Real-Time Coordination Pattern. Channel delays and reliability are both of crucial importance to the real-time coordination patterns. We address them explicitly by giving one RTSC for each connector. A real-time pattern is then formally defined as follows:

Definition 7. *A* real-time coordination pattern *(collaboration type) $\mathcal{P}$ is a tuple $(R_{\mathcal{P}}, \Psi_{\mathcal{P}}, \phi_{\mathcal{P}}, \mathcal{C}_{\mathcal{P}})$ with $R_{\mathcal{P}}$ a set of roles $(r_i, \mathbb{M}^{r_i}_{\mathcal{P}})$ for $r_i \in \mathcal{N}_{\mathcal{R}}$ a role name and $\mathbb{M}^{r_i}_{\mathcal{P}}$ a role behavior in the form of a RTSC, a set $\Psi_{\mathcal{P}}$ of OCL invariants $\psi_1, \ldots,$ ψ_k for each role, the RT-OCL pattern constraint $\phi_{\mathcal{P}}$, and the atomic component type $\mathcal{C}_{\mathcal{P}}$ representing the connectors.*[5]

For connector instance (collaboration instances) $o \in \mathcal{N}^+_{\mathcal{C}}$ of the pattern $\mathcal{P}$ we refer to the pattern constraints as $\phi^o_{\mathcal{P}}$ and to the related *behavior* as $\mathbb{M}^o_{\mathcal{P}}$ which is derived from $\mathbb{M}_{\mathcal{P}}$ for $\mathcal{C}_{\mathcal{P}} = (\mathcal{S}_{\mathcal{P}}, \mathbb{M}_{\mathcal{P}})$ by renaming the ports accordingly.

Agents. To capture agents which realize the peer-to-peer interaction of autonomous mechatronic systems as well as the embedding of complex hierarchies (cf. Fig. 2), we define an agent as a special component as follows:

Definition 8. *An* agent *is a reconfigurable hierarchical component type $\mathcal{A} = (\mathcal{S}_{\mathcal{A}}, \mathbb{M}_{\mathcal{A}}, \phi_{\mathcal{A}})$ with $\mathcal{S}_{\mathcal{A}} = (\mathcal{I}_{\mathcal{A}}, \mathcal{E}_{\mathcal{A}}, map_{\mathcal{A}})$ such that the interface is mode-independent ($\mathcal{I}_{\mathcal{A}}$ is constant) and the internal behavior $\mathbb{M}_{\mathcal{A}}$ is decomposed into $\mathbb{M}^s_{\mathcal{A}} \| \mathbb{M}^{p_1}_{\mathcal{A}} \| \ldots \| \mathbb{M}^{p_h}_{\mathcal{A}}$ where $\mathbb{M}^{p_i}_{\mathcal{A}}$ refines the port behavior $\mathbb{M}^{p_i}_i$ for $\mathcal{I}_{\mathcal{A}} = \{(p_1, \mathbb{M}_1), \ldots, (p_h, \mathbb{M}_h)\}$.*

The ports of an agent are assumed either to be not considered within our behavioral model (unsafe ports) or related to a specific role of a pattern instance (regular port). The set of associated OCL role invariants $\psi_1, \ldots, \psi_h$ of $\mathcal{A}$ by $\Psi_{\mathcal{A}}$ and the resulting overall component OCL role invariant $\psi_{\mathcal{A}}$ is therefore derived by combining the related OCL role invariants $(\psi_1 \wedge \cdots \wedge \psi_h)$. For agent instances $o \in \mathcal{N}^+_{\mathcal{C}}$ we refer to the component OCL role invariant as $\psi^o_{\mathcal{A}}$. A hierarchical system with an agent as top level component type is further named *agent subsystem.*

[5] A real-time pattern is *consistent* if the roles cover the complete interaction which is possible via the component representing the related connectors (see [3]).

Peer-to-Peer System. Peer-to-peer systems are built by composing atomic agents via pattern instances as depicted in Fig. 11. The composition of atomic agents via pattern instances depicted in Fig. 11 as upper part of the core can be formally defined as follows:

Definition 9. *A peer-to-peer system $\mathfrak{S}^p$ is a tuple $(O^c_{\mathfrak{S}^p}, O^p_{\mathfrak{S}^p}, c_{\mathfrak{S}^p}, p_{\mathfrak{S}^p}, map_{\mathfrak{S}^p})$ with $O^c_{\mathfrak{S}^p} \subseteq \wp(\mathcal{N}_\mathcal{C})$ a set of instance names of atomic agents $c_1, \dots, c_n$, $O^p_{\mathfrak{S}^p} \subseteq \wp(\mathcal{N}_\mathcal{C})$ a set of instance names of the connector components representing patterns $p_1, \dots, p_m$ with $O^c_{\mathfrak{S}^p} \cap O^p_{\mathfrak{S}^p} = \emptyset$, $c_{\mathfrak{S}^p}$ a function which maps to each instance $c_i \in O^c_{\mathfrak{S}^p}$ a related component type, $p_{\mathfrak{S}^p}$ a function which maps to each instance $p_j \in O^p_{\mathfrak{S}^p}$ the related pattern, and $map_{\mathfrak{S}^p} : (O_p.\mathcal{N}_\mathcal{Q}) \to ((O_c \cap \mathcal{N}_\mathcal{C}).\mathcal{N}_\mathcal{Q})$ a bijective mapping which connects ports of components representing the pattern connectors with the ports of the root components of agents.*

For such a peer-to-peer system $\mathfrak{S}^p = (O^c_{\mathfrak{S}^p}, O^p_{\mathfrak{S}^p}, c_{\mathfrak{S}^p}, p_{\mathfrak{S}^p}, map_{\mathfrak{S}^p})$ we have to combine the behavior related to the agent instances and pattern instances. The overall behavior and safety property of a system as depicted in Fig. 11 as upper part of the core is given by

$$\mathbb{M}_{\mathfrak{S}^p} := \left(\mathop{\|}_{c \in O^c_{\mathfrak{S}^p}} \mathbb{M}^c_{c_{\mathfrak{S}^p}(c)} \right) \| \left(\mathop{\|}_{p \in O^p_{\mathfrak{S}^p}} \mathbb{M}^p_{p_{\mathfrak{S}^p}(p)} \right) \qquad \phi_{\mathfrak{S}^p} := \bigwedge_{c \in O^c_{\mathfrak{S}^p}} \phi^c_{c_{\mathfrak{S}^p}(c)}.$$

The safety of a peer-to-peer system can then be formally defined referring to the overall behavior, pattern constraints, and component invariants as follows (by providing a formal version for Definition 2):

Definition 10. *A peer-to-peer system $\mathfrak{S}^p = (O^c_{\mathfrak{S}^p}, O^p_{\mathfrak{S}^p}, c_{\mathfrak{S}^p}, p_{\mathfrak{S}^p}, map_{\mathfrak{S}^p})$ is* safe *if the following conditions are fulfilled by the behavior $\mathbb{M}_{\mathfrak{S}^p}$:*

– *all agents are locally safe and no deadlock occurs:* $\mathbb{M}_{\mathfrak{S}^p} \models \phi_{\mathfrak{S}^p} \wedge \neg\delta$ (9)

– *All RT-OCL constraints of patterns are fulfilled:* $\mathbb{M}_{\mathfrak{S}^p} \models \wedge_{o \in O^p_{\mathfrak{S}^p}} \phi^o$ (10)

– *All OCL role invariants of the agents are fulfilled:* $\mathbb{M}_{\mathfrak{S}^p} \models \wedge_{o \in O^c_{\mathfrak{S}^p}} \psi^o$. (11)

Compositional Verification. To guarantee that the peer-to-peer real-time coordination of the whole system is safe, we have to look locally at the patterns and the role refinement by the agents (cf. Fig. 16) before we can compose these results for the peer-to-peer coordination.

Pattern Verification. We verify whether the behavioral requirement in form of safety properties specified by means of RT-OCL hold for a real-time pattern. If the requirement is fulfilled, the pattern is locally safe. Formally, a real-time pattern $\mathcal{P} = (R_\mathcal{P}, \Psi_\mathcal{P}, \phi_\mathcal{P}, \mathcal{C}_\mathcal{P})$ with a set R of roles with a name and a RTSCs for

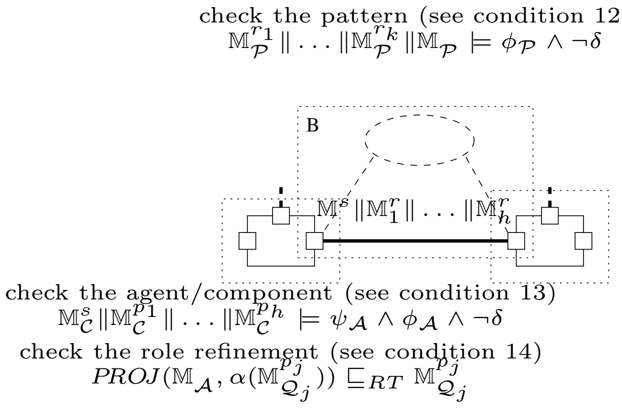

check the pattern (see condition 12)
$\mathbb{M}_{\mathcal{P}}^{r1} \| \dots \| \mathbb{M}_{\mathcal{P}}^{rk} \| \mathbb{M}_{\mathcal{P}} \models \phi_{\mathcal{P}} \wedge \neg\delta$
B
$\mathbb{M}^s \| \mathbb{M}_1^r \| \dots \| \mathbb{M}_h^r$
check the agent/component (see condition 13)
$\mathbb{M}_{\mathcal{C}}^s \| \mathbb{M}_{\mathcal{C}}^{p1} \| \dots \| \mathbb{M}_{\mathcal{C}}^{ph} \models \psi_{\mathcal{A}} \wedge \phi_{\mathcal{A}} \wedge \neg\delta$
check the role refinement (see condition 14)
$PROJ(\mathbb{M}_{\mathcal{A}}, \alpha(\mathbb{M}_{\mathcal{Q}_j}^{p_j})) \sqsubseteq_{RT} \mathbb{M}_{\mathcal{Q}_j}^{p_j}$

Fig. 16. Verification of peer-to-peer structures via patterns and role refinement

each role $(r_1, \mathbb{M}_{\mathcal{P}}^{r_1}), \dots, (r_k, \mathbb{M}_{\mathcal{P}}^{r_k})$ and behavior $\mathbb{M}_{\mathcal{P}}$ for the connector component $\mathcal{C}_{\mathcal{P}} = (\mathcal{S}_{\mathcal{P}}, \mathbb{M}_{\mathcal{P}})$ is a *locally safe* real-time pattern if:

$$\mathbb{M}_{\mathcal{P}}^{r_1} \| \dots \| \mathbb{M}_{\mathcal{P}}^{r_k} \| \mathbb{M}_{\mathcal{P}} \models \phi_{\mathcal{P}} \wedge \neg\delta. \tag{12}$$

The behavior $\mathbb{M}_{\mathcal{P}}^{r_1} \| \dots \| \mathbb{M}_{\mathcal{P}}^{r_k} \| \mathbb{M}_{\mathcal{P}}$ is supposed to be a closed real-time behavior and can thus be verified using a real-time model checker for RTSC by checking whether the constraint $\phi \wedge \neg\delta$ holds.

In our example we generate model checker input from the RTSC for FrontRole, RearRole and an additional RTSC for the implicitly defined connector.

Agent Verification. Besides the patterns also the agents have to be verified. We have to verify whether the agent behavior respects the role RTSC and the role invariants defined as local safety of the agent.

An agent $\mathcal{A} = (\mathcal{S}_{\mathcal{A}}, \mathbb{M}_{\mathcal{A}}, \phi_{\mathcal{A}})$ with internal behavior $\mathbb{M}_{\mathcal{A}}$ can be decomposed into $\mathbb{M}_{\mathcal{C}}^s \| \mathbb{M}_{\mathcal{C}_1}^{p_1} \| \dots \| \mathbb{M}_{\mathcal{C}_h}^{p_h}$. The RTSCs $\mathbb{M}_{\mathcal{C}_1}^{p_1}, \dots, \mathbb{M}_{\mathcal{C}_h}^{p_h}$ have to refine the port behavior $\mathbb{M}_{\mathcal{Q}_1}^{p_1}, \dots, \mathbb{M}_{\mathcal{Q}_h}^{p_h}$ for $\mathcal{I}_{\mathcal{A}} = \{(p_1, \mathbb{M}_{\mathcal{Q}_1}), \dots, (p_h, \mathbb{M}_{\mathcal{Q}_h})\}$ and the HRTSC $\mathbb{M}_{\mathcal{C}}^s$ describes the component internal synchronization, and the reconfiguration and embedding of subordinated hybrid reconfigurable components. Such an agent with agent OCL role invariant $\psi_{\mathcal{A}}$ the is *locally safe* if:

$$\mathbb{M}_{\mathcal{C}}^s \| \mathbb{M}_{\mathcal{C}_1}^{p_1} \| \dots \| \mathbb{M}_{\mathcal{C}_h}^{p_h} \models \psi_{\mathcal{A}} \wedge \phi_{\mathcal{A}} \wedge \neg\delta \tag{13}$$

Using the related RTSC $RT(\mathbb{M}_{\mathcal{C}}^s)$ instead of the HRTSC $\mathbb{M}_{\mathcal{C}}^s$ we can use a real-time model checker to prove $\psi_{\mathcal{A}} \wedge \phi_{\mathcal{A}} \wedge \neg\delta$. As $\psi_{\mathcal{A}} \wedge \phi_{\mathcal{A}} \wedge \neg\delta$ does not refer to any continuous variables which are not clock variables, the verification result for $RT(\mathbb{M}_{\mathcal{C}}^s \| \mathbb{M}_{\mathcal{C}_1}^{p_1} \| \dots \| \mathbb{M}_{\mathcal{C}_h}^{p_h})$ also holds for $\mathbb{M}_{\mathcal{C}}^s \| \mathbb{M}_{\mathcal{C}_1}^{p_1} \| \dots \| \mathbb{M}_{\mathcal{C}_h}^{p_h}$ if the embedding is safe (cf. Section 4.1). Note that, as $RT(\mathbb{M}_{\mathcal{C}}^s) \| \mathbb{M}_{\mathcal{C}_1}^{p_1} \| \dots \| \mathbb{M}_{\mathcal{C}_h}^{p_h}$ is an open model, we assume the erratic but guaranteed execution of external signals when performing the model checking as outlined in [14].

In our example the invariant for the shuttle component is automatically derived from the role invariants ψ_1 and ψ_2 (see constraints (1) and (2)). In the resulting invariant, frontRole and rearRole are now specific names for navigation to the associated ports. The same abstract shuttle property CanBrakeFully is replaced by the or-combination of all states of the synchronization chart which fulfill them: not self.CanBrakeFully equals Synchronization::convoyFront) and self.CanBrakeFully equals Synchronization::convoyFront or Synchronization::convoyyRear or shorter not Synchronization::convoyFront).

```
context Shuttle inv:
 (oclInState(frontRole::convoy) implies oclInState(Synchronization::convoyFront)) and
 (oclInState(rearRole::convoy) implies not oclInState(Synchronization::convoyFront))
```

Regular Ports. Each port RTSC M_j^r of the agent $\mathcal{A}$ has to refine the port behavior $\mathbb{M}_{\mathcal{Q}_j}^{p_j}$ for $\mathcal{I}_{\mathcal{A}} = \{(p_1, \mathbb{M}_1), \dots, (p_h, \mathbb{M}_h)\}$ which is equal to the connected pattern role behavior. The whole agent behavior $\mathbb{M}_{\mathcal{A}}$ restricted to the interface of the port $(p_j, \mathbb{M}_j)$ has to result in such a refinement.

$$\mathit{PROJ}(\mathbb{M}_{\mathcal{A}}, \alpha(\mathbb{M}_{\mathcal{Q}_j}^{p_j})) \sqsubseteq_{RT} \mathbb{M}_{\mathcal{Q}_j}^{p_j}. \tag{14}$$

where $\mathit{PROJ}(\mathbb{M}, A)$ denotes the automaton which results when all transitions with input and output signals not present in A are replaced by non-deterministic ones (cf. [14]) for an automaton $\mathbb{M}$ and a given set of labels A.

To ensure that $\mathbb{M}_{\mathcal{A}}$ refines each of the role protocols associated to its ports, we propose the use of syntactical refinement rules which ensure $\mathit{PROJ}(\mathbb{M}_{\mathcal{A}}, \alpha(\mathbb{M}_{\mathcal{Q}_j}^{p_j})) \sqsubseteq_{RT} \mathbb{M}_{\mathcal{Q}_j}^{p_j}$. Requiring disjoint signal labels and checking in addition $\mathbb{M}_{\mathcal{A}} \models \neg\delta$, we can then ensure that condition 14 holds. Alternatively, model checking can be employed to also fully automate this task (cf. [31–33]).

The RTSC in Fig. 12 is a refinement of the roles from Fig. 9. Consequently, it needs to be ensured that the embedding of AC only refines the specified real-time behavior from Fig. 12 and does not add additional behavior, or be in conflict with the real-time specification of this super-ordinated component.

Unsafe Ports. In order to be able to verify partial systems, we have initially introduced the classification *unsafe* ports. For these unsafe ports proper decoupling but no refinement has to be checked to ensure safety. The idea includes two steps. (1) When checking condition 13 for the component type that has an unsafe port in the case of either a top level or embedded component instance, we simply consider the transitions that interact with an unsafe port to occur erratic and non-urgent. (2) An additional verification step proves that the unsafe port cannot block the component behavior. We therefore use a function *NDET* to transform the RTSC of the port into another one where all external communication is replaced by purely erratic non-deterministic behavior.

To ensure safety we then have to check that the unsafe role RTSC $\mathbb{M}_{\mathcal{Q}_j}^{p_j}$ transformed by *NDET* is deadlock free so that the component can never be blocked via this port.

$$\mathit{NDET}(\mathbb{M}_{\mathcal{Q}_j}^{p_j}) \models \neg\delta \tag{15}$$

In our example the *decoupled interaction* designed in Fig. 13 results in an unsafe port. The resulting check transforms the role RTSC using *NDET* and then checks the resulting RTSC for deadlocks.

Peer to Peer Model Verification. These separate results can be composed using a compositional reasoning scheme to conclude that the peer-to-peer coordination is safe (as outlined in [7] for the case without unsafe ports).

Theorem 2. *A consistent* peer-to-peer system $\mathfrak{S}^c = (O^c_{\mathfrak{S}^c}, O^p_{\mathfrak{S}^c}, c_{\mathfrak{S}^c}, p_{\mathfrak{S}^c}, map_{\mathfrak{S}^c})$ *is* safe *if*

- *all patterns are locally safe (condition 12),*
- *all agents/components are locally safe (condition 13),*
- *all ports are behavioral consistent (condition 14), and*
- *all unsafe ports are behavioral consistent (condition 15).*

Proof. (sketch) As we restricted the RT-OCL constraints to compositional properties and only considered the real-time behavior of the top-level components, we can use the border built by the ports and roles to also prove the constraints $\phi_{\mathcal{P}}$ and invariants $\psi_{\mathcal{C}_j}$ compositionally. We therefore use the local checks for the real-time patterns and top-level hybrid components and the refinement $\mathbb{M}_{\mathcal{A}} \sqsubseteq_{RT} \mathbb{M}^{p_1}_{\mathcal{Q}_1} \| \dots \| \mathbb{M}^{p_h}_{\mathcal{Q}_h}$ (cf. [7]). The advantage of the compositional approach is that it permits us to verify condition 16, 17, 18 and 19 without building the state space for $\mathbb{M}_{\mathfrak{S}^c}$. Instead, only the consistency of the overall system, the local safety for all patterns, components, and the proper behavioral consistency of the ports concerning the fulfilled roles has to be ensured.

If unsafe ports are also present in the safety-critical subsystem, the check $NDET(\mathbb{M}^{p_j}_{\mathcal{Q}_j}) \models \neg\delta$ guarantees that the remaining uncovered environment cannot invalidate the result which has been achieved for the verified ones, even though they may not conform to behavior specified by the pattern roles.[6]

For the peer-to-peer interaction, we employ model checking to check conditions 12, 13, 14, and 15. The compositional reasoning sketched in Theorem 2 proves that these local checks result in guarantees for the overall system.

It the next section we will extend this result to systems with hierarchical embedding.

4.3 Overall Model

Syntax, Semantics and Safety. Our approach for the modeling of the safety-critical core of a hierarchy of OCM, as depicted in Fig. 2, is based on the observation that it can be formally defined by a set of hierarchical agents and pattern instances (Fig. 11). While the free peer-to-peer interaction of the top-level OCMs can be captured by pattern instances, a hierarchical system of configurable components can be used to cover the hierarchies of reflective operators and controllers.

[6] We assume that no invariants and constraints for the un-verified patterns and components exist and the related elements in the formal model are set to true.

The outlined composition of agents via pattern instances to build the safety-critical core can be formally defined by combining Definition 5 and 9 as follows:

Definition 11. *A* core system $\mathfrak{S}^c$ *is a tuple* $(O^c_{\mathfrak{S}^c}, O^p_{\mathfrak{S}^c}, c_{\mathfrak{S}^c}, p_{\mathfrak{S}^c}, map_{\mathfrak{S}^c})$ *with* $O^c_{\mathfrak{S}^c} \subseteq \wp(\mathcal{N}^+_{\mathcal{C}})$ *a set of* n *names of agents relating to the hierarchial systems* $(O^c_{\mathfrak{S}^c} = O^c_1 \uplus \cdots \uplus O^c_n$ *and all* $\mathfrak{S}^h_i = (O^c_i, c_{\mathfrak{S}^c}|_{O^c_i})$ *are hierarchical systems),* $O^p_{\mathfrak{S}^c} \subseteq \wp(\mathcal{N}_{\mathcal{C}})$ *a set of instance names* $p_1, \dots, p_m$ *of the connector components representing patterns with* $O^c_{\mathfrak{S}^c} \cap O^p_{\mathfrak{S}^c} = \emptyset$*,* $c_{\mathfrak{S}^c}$ *a function which maps to each instance* $c_i \in O^c_{\mathfrak{S}^c}$ *a related component type,* $p_{\mathfrak{S}^c}$ *a function which maps to each instance* $p_j \in O^p_{\mathfrak{S}^c}$ *the related pattern, and* $map_{\mathfrak{S}^c} : (O_p.\mathcal{N}_{\mathcal{Q}}) \to ((O_c \cap \mathcal{N}_{\mathcal{C}}).\mathcal{N}_{\mathcal{Q}})$ *a bijective mapping which connects ports of components representing the pattern connectors with the ports of the root components of agents.*

The remaining part of the architecture consisting of the cognitive operators, other components outside the core, and their interconnections is also covered by related components whose connection with the safety-critical core are only unsafe ports (cf. Fig. 11).

To cover the complete system, including the cognitive operators, we employ the following extension:

Definition 12. *A* system $\mathfrak{S}$ *is a tuple* $(O^c_{\mathfrak{S}}, O^p_{\mathfrak{S}}, c_{\mathfrak{S}}, p_{\mathfrak{S}}, map_{\mathfrak{S}})$ *which includes a core system* $\mathfrak{S}^c = (O^c_{\mathfrak{S}^c}, O^p_{\mathfrak{S}^c}, c_{\mathfrak{S}^c}, p_{\mathfrak{S}^c}, map_{\mathfrak{S}^c})$.[7]

For a system $\mathfrak{S}$ and core $\mathfrak{S}^c$ we have to combine the behavior related to the component instances and pattern instances to get the overall behavior:

$$\mathbb{M}_{\mathfrak{S}} := \left(\mathop{\|}_{c \in O^c_{\mathfrak{S}}} \mathbb{M}^c_{c_{\mathfrak{S}}(o_p)} \right) \| \left(\mathop{\|}_{p \in O^p_{\mathfrak{S}}} \mathbb{M}^p_{p_{\mathfrak{S}}(o_p)} \right) \quad \mathbb{M}_{\mathfrak{S}^c} := \left(\mathop{\|}_{c \in O^c_{\mathfrak{S}^c}} \mathbb{M}^c_{c_{\mathfrak{S}^c}(p)} \right) \| \left(\mathop{\|}_{p \in O^p_{\mathfrak{S}^c}} \mathbb{M}^p_{p_{\mathfrak{S}^c}(p)} \right).$$

In our example we have the top-level component Shuttle which is connected via *map* with instances of the ConvoyCoordination pattern. A single supervised embedded component instance of type AC exists for each Shuttle instance.

The overall safety of a system combining Definition 6 and 10 can then be defined referring to the overall behavior, pattern constraints, and component invariants as follows (by providing a formal version for Definition 3):

Definition 13. *A* system $\mathfrak{S} = (O^c_{\mathfrak{S}}, O^p_{\mathfrak{S}}, c_{\mathfrak{S}}, p_{\mathfrak{S}}, map_{\mathfrak{S}})$ *with included core system* $\mathfrak{S}^c = (O^c_{\mathfrak{S}^c}, O^p_{\mathfrak{S}^c}, c_{\mathfrak{S}^c}, p_{\mathfrak{S}^c}, map_{\mathfrak{S}^c})$ *is* safe *if the following conditions for the behavior are fulfilled:*

– *Local safety is fulfilled for the core:* $\mathbb{M}_{\mathfrak{S}^c} \models \wedge_{c \in O^c_{\mathfrak{S}^c}} \phi^c_{c_{\mathfrak{S}^c}(c)}$ (16)

– *Deadlock freedom is guaranteed for the core:* $\mathbb{M}_{\mathfrak{S}^c} \models \neg\delta$ (17)

– *All RT-OCL constraints for patterns are fulfilled:* $\mathbb{M}_{\mathfrak{S}} \models \wedge_{o \in O^p_{\mathfrak{S}^c}} \phi^o$ (18)

[7] Inclusion of a system in another system is defined formally in [3].

– *All OCL role invariants of agents are fulfilled:* $\mathbb{M}_{\mathfrak{S}} \models \wedge_{o \in O^c_{\mathfrak{S}^c}} \psi^o_{c_{\mathfrak{S}^c}(c)}$ (19)

Condition 17 ensure that the liveness properties defined in the role and port protocols are guaranteed by the core behavior, while condition 16, 18 and 19 ensure that the safety properties of the patterns and involved agents/components in the core are fulfilled by the overall behavior.

Overall Verification. For the decomposition of a system into a safety-critical core and the rest as depicted in Fig. 11 we now show that the safety of the core is not affected when composed with an arbitrary rest system.

Theorem 3. *Any system* $\mathfrak{S} = (O^c_{\mathfrak{S}}, O^p_{\mathfrak{S}}, c_{\mathfrak{S}}, p_{\mathfrak{S}}, map_{\mathfrak{S}})$ *with included core system* $\mathfrak{S}^c = (O^c_{\mathfrak{S}^c}, O^p_{\mathfrak{S}^c}, c_{\mathfrak{S}^c}, p_{\mathfrak{S}^c}, map_{\mathfrak{S}^c})$ *is safe if the core* $\mathfrak{S}^c$ *is safe.*

Proof. (sketch) Conditions 16 and 17 obviously holds as only the core is considered and the safety of the core guarantees it. Due to the checks for the unsafe ports, condition 18 and 19 can also be preserved when composing the core with the rest.

The following Corollary summarizes that the outlined separate results for the compositional and modular verification of MUML models can be combined to ensure the safety of the overall system (as defined in Definition 13).

Corrollary 1. *A* system $\mathfrak{S} = (O^c_{\mathfrak{S}}, O^p_{\mathfrak{S}}, c_{\mathfrak{S}}, p_{\mathfrak{S}}, map_{\mathfrak{S}})$ *with included core system* $\mathfrak{S}^c = (O^c_{\mathfrak{S}^c}, O^p_{\mathfrak{S}^c}, c_{\mathfrak{S}^c}, p_{\mathfrak{S}^c}, map_{\mathfrak{S}^c})$ *is* safe *if*

– *all embedded component types are locally safe (condition 3 and 5),*
– *all embeddings are behaviorally consistent (see condition 4 or 6),*
– *all patterns are locally safe (condition 12),*
– *all agents are locally safe (condition 13),*
– *all regular ports are behaviorally consistent (condition 14), and*
– *all unsafe ports are behaviorally consistent (condition 15).*

Proof. (sketch) For the non-hierarchial case the result follows from Theorem 2. To also take the hierarchical embedding of subordinated components into account, we refer to Theorem 1 which guarantees that the whole behavior is well-formed and that the real-time behavior of the top-level components is always only refined by the behavioral consistent embeddings. The safety of the core system is not affected by the rest of the system following Theorem 3.

5 Related Work

The UML concepts themselves without the MUML refinements and UML extension for real-time, such as the UML Profile for Modeling and Analysis of Real-Time Embedded Systems (MARTE) [35] and its extension MARTE-DAM [36] for dependability analysis are not sufficient for the model-driven development of advanced mechatronic systems as targeted in the paper. They are neither defined rigorously enough to support a decomposition for the analysis nor appropriately tailored to support systems with self-optimization. The System Modeling Language (SysML) [37], which combines UML concepts with system engineering concepts has the same limitations and restricts its attention to requirement and early phases and thus does not provide the required support for the later phases.

The Koala Component Model for Consumer Electronics Software [38] is one example where reconfiguration has been taken into account in a similar setting. However, the model is restricted to the component structure only and does not cover real-time or hybrid behavior.

In the OMEGA project [39], the UML has been extended by additional time constructs. However, in contrast to our approach, there is no support for hybrid behavior, and compositional verification is only supported by semi-automatic verification via theorem proving.

The description of control algorithms by time-continuous variables and corresponding ports is similar to other approaches such as HyROOM [40] and the underlying HyCharts [41]. Masaccio [42] and CHARON [28,43,44] also support the component-based modeling of hybrid systems and verification. The software's architecture is specified similarly to ROOM/UML-RT and the behavior is specified by statecharts whose states are associated with systems of ordinary differential equations, differential constraints or Matlab/Simulink block diagrams. These approaches provide means for the reconfiguration of systems in terms of changing the continuous behavior. However, it is only possible to reconfigure the model inside a component on one hierarchy-level. Our approach allows for a complex reconfiguration altering the structure across more than one hierarchy-level. For a more comprehensive comparison of a number of modeling techniques for advanced mechatronic systems, which also addresses the adaptation aspect, we refer to [45].

Concerning the verification of adaptive behavior, only first attempts exist. In [46], as in our presented work, verification techniques are employed to ensure that the self-adaptive behavior does not result in any harm. We additionally include self-coordinating behavior, suggest a compositional approach which can also be employed to study systems unable to be addressed as a whole. We also take real-time and continuous behavior into account. In [47], required properties for untimed models are checked under the assumption that a self-x capability of a system will fix certain types of problems in the long run. Also, we have to provide guarantees for the self-optimizing mechatronic systems in all possible cases under hard real-time constraints. Due to the multiple involved agents and their limitations (only local knowledge and limited reasoning capabilities) we cannot rely on the adaptation capabilities of the agents. Another direction for assurance is runtime verification [48]. However, it would be too late for the considered class of safety guarantees if the problems are detected at runtime. Only in cases where the runtime verification is not required in hard real-time (and thus remain outside the safe core) such an approach seems reasonable. A possible orthogonal extension of the presented approach is therefore to perform such runtime verification steps in the cognitive operator.

6 Conclusion

The MUML approach enables the model-driven development of mechatronic systems with advanced capabilities such as self-adaptive run-time behavior by providing the following three building blocks: (1) A suitable modeling approach for hierarchical structures of OCMs is provided which supports the specification of hybrid behavior and the reconfiguration of subsystems in order to support the reliable self-adaptation of the OCMs. (2) For the level of freely interacting software agents, the flexible but safe real-time coordination between the autonomous mechatronic agents is achieved employing the coordination pattern concept. (3) The approach integrates the hierarchical OCM structures and flexible interaction of software agents in such a manner that safety properties can be verified based on compositional checking so that the approach becomes scalable.

Acknowledgements. We thank Christian Heinzemann and Stefan Henkler for their comments on earlier versions of this paper. We are also grateful to all the other PhD students and students who worked on MUML or implemented the tool support for the described concepts as extensions for the Fujaba UML CASE tool.

References

1. Schäfer, W., Wehrheim, H.: The challenges of building advanced mechatronic systems. In: FOSE 2007: 2007 Future of Software Engineering, pp. 72–84. IEEE Computer Society, Washington (2007)
2. Sztipanovits, J., Karsai, G., Bapty, T.: Self-adaptive software for signal processing. Commun. ACM 41(5), 66–73 (1998)
3. Giese, H., Schäfer, W.: Model-driven development of safe self-optimizing mechatronic systems with mechatronic uml. Technical Report tr-ri-12-322, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Paderborn, Germany (2012), `http://www.cs.uni-paderborn.de/uploads/tx_sibibtex/GS12.pdf`
4. Burmester, S., Tichy, M., Giese, H.: Modeling Reconfigurable Mechatronic Systems with Mechatronic UML. In: Aßmann, U. (ed.) Proc. of Model Driven Architecture: Foundations and Applications (MDAFA 2004), Linköping, Sweden, pp. 155–169 (June 2004)
5. Burmester, S., Giese, H., Tichy, M.: Model-Driven Development of Reconfigurable M. In: Aßmann, U., Akşit, M., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 47–61. Springer, Heidelberg (2005)
6. Giese, H.: A Formal Calculus for the Compositional Pattern-Based Design of Correct Real-Time Systems. Technical Report tr-ri-03-240, Lehrstuhl für Softwaretechnik, Universität Paderborn, Paderborn, Deutschland (July 2003)
7. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the Compositional Verification of Real-Time UML Designs. In: Proc. of the 9th European Software Engineering Conference held Jointly with 11th ACM SIGSOFT international Symposium on Foundations of Software Engineering (ESEC/FSE 2011), pp. 38–47. ACM Press (September 2003)

8. Burmester, S., Giese, H., Oberschelp, O.: Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In: Araujo, H., Vieira, A., Braz, J., Encarnacao, B., Carvalho, M. (eds.) Proc. of 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004), Setubal, Portugal, pp. 222–229. INSTICC Press (August 2004)
9. Giese, H., Burmester, S., Schäfer, W., Oberschelp, O.: Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In: Roy, B., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017, pp. 179–188. Springer, Heidelberg (2004)
10. Burmester, S., Giese, H., Oberschelp, O.: Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In: Informatics in Control, Automation and Robotics. Kluwer Academic Publishers, Dordrecht (2005)
11. Hestermeyer, T., Oberschelp, O., Giese, H.: Structured Information Processing For Self-optimizing Mechatronic Systems. In: Araujo, H., Vieira, A., Braz, J., Encarnacao, B., Carvalho, M. (eds.) Proc. of 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004), pp. 230–237. INSTICC Press, Setubal (2004)
12. 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)
13. Burmester, S., Giese, H., Münch, E., Oberschelp, O., Klein, F., Scheideler, P.: Tool Support for the Design of Self-Optimizing Mechatronic Multi-Agent Systems. International Journal on Software Tools for Technology Transfer (STTT) 10(3), 207–222 (2008)
14. Burmester, S., Giese, H., Hirsch, M., Schilling, D.: Incremental design and formal verification with UML/RT in the FUJABA real-time tool suite. In: Proc. of the International Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML 2004, pp. 1–20 (October 2004)
15. Burmester, S., Giese, H., Hirsch, M., Schilling, D., Tichy, M.: The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. In: Proc. of the 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, USA (May 2005)
16. Burmester, S., Giese, H., Schäfer, W.: Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 25–40. Springer, Heidelberg (2005)
17. Burmester, S., Giese, H., Gambuzza, A., Oberschelp, O.: Partitioning and Modular Code Synthesis for Reconfigurable Mechatronic Software Components. In: Bobeanu, C. (ed.) Proc. of European Simulation and Modelling Conference (ESMc 2004), Paris, France, pp. 66–73. EOROSIS Publications, Paris (2004)
18. Giese, H., Henkler, S., Hirsch, M.: A multi-paradigm approach supporting the modular execution of reconfigurable hybrid systems. Simulation 87(9), 775–808 (2011)
19. Oberschelp, O., Gambuzza, A., Burmester, S., Giese, H.: Modular Generation and Simulation of Mechatronic Systems. In: Proc. of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics, SCI, Orlando, USA (July 2004)

20. Heinzemann, C., Pohlmann, U., Rieke, J., Schäfer, W., Sudmann, O., Tichy, M.: Generating simulink and stateflow models from software specifications. In: Proceedings of the 12th International Design Conference, DESIGN 2012 (May 2012) (accepted)
21. Giese, H., Burmester, S.: Real-Time Statechart Semantics. Technical Report tr-ri-03-239, Lehrstuhl für Softwaretechnik, Universität Paderborn, Paderborn, Germany (June 2003)
22. Burmester, S., Giese, H.: The Fujaba Real-Time Statechart PlugIn. In Giese, H., Zündorf, A., eds.: Proc. of the first International Fujaba Days 2003, Kassel, Germany. Volume tr-ri-04-247 of Technical Report., pp. 1–8. University of Paderborn (October 2003)
23. Larsen, K., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. Springer International Journal of Software Tools for Technology 1(1) (1997)
24. Henzinger, T.A., Manna, Z., Pnueli, A.: What Good Are Digital Clocks? In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992)
25. OMG: UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-03-02 (September 2002)
26. Henzinger, T.A., Ho, P.H., Wong-Toi, H.: HyTech: The Next Generation. In: Proc. of the 16th IEEE Real-Time Symposium. IEEE Computer Press (December 1995)
27. Bender, K., Broy, M., Peter, I., Pretschner, A., Stauner, T.: Model based development of hybrid systems. In: Modelling, Analysis, and Design of Hybrid Systems. LNCIS, vol. 279, pp. 37–52. Springer, Heidelberg (2002)
28. Alur, R., Dang, T., Esposito, J., Fierro, R., Hur, Y., Ivancic, F., Kumar, V., Lee, I., Mishra, P., Pappas, G., Sokolsky, O.: Hierarchical Hybrid Modeling of Embedded Systems. In: First Workshop on Embedded Software (2001)
29. Lynch, N.A.: Input/Output Automata: Basic, Timed, Hybrid, Probabilistic, Dynamic,.. In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 191–192. Springer, Heidelberg (2003)
30. Flake, S., Mueller, W.: An OCL Extension for Real-Time Constraints. In: Clark, A., Warmer, J. (eds.) Object Modeling with the OCL. LNCS, vol. 2263, pp. 150–171. Springer, Heidelberg (2002)
31. Giese, H., Hirsch, M.: Modular Verification of Safe Online-Reconfiguration for Proactive Components in Mechatronic UML. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 67–78. Springer, Heidelberg (2006)
32. Giese, H., Hirsch, M.: Modular Verification of Safe Online-Reconfiguration for Proactive Components in Mechatronic UML. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 67–78. Springer, Heidelberg (2006)
33. Giese, H., Hirsch, M.: Checking and Automatic Abstraction for Timed and Hybrid Refinement in Mechtronic UML. Technical Report tr-ri-03-266, University of Paderborn, Paderborn, Germany (December 2005)
34. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? Journal of Computer and System Sciences 57, 94–124 (1998); A preliminary version appeared in the Proceedings of the 27th Annual Symposium on Theory of Computing (STOC), pp. 373–382. ACM Press (1995)
35. OMG: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Version 1.1 (June 2011)
36. Bernardi, S., Merseguer, J., Petriu, D.C.: A dependability profile within MARTE. Softw. Syst. Model. 10(3), 313–336 (2011)
37. Object Management Group: Systems Modeling Language (SysML) Specification (January 2005)

38. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The koala component model for consumer electronics software. Computer 33(3), 78–85 (2000)
39. Graf, S., Hooman, J.: Correct Development of Embedded Systems. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, pp. 241–249. Springer, Heidelberg (2004)
40. Stauner, T., Pretschner, A., Péter, I.: Approaching a Discrete-Continuous UML: Tool Support and Formalization. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 242–257. Springer, Heidelberg (2001)
41. Stauner, T.: Systematic Development of Hybrid Systems. PhD thesis, Technische Universität München (2001)
42. Henzinger, T.A.: Masaccio: A Formal Model for Embedded Components. In: Watanabe, O., Hagiya, M., Ito, T., van Leeuwen, J., Mosses, P.D. (eds.) TCS 2000. LNCS, vol. 1872, pp. 549–563. Springer, Heidelberg (2000)
43. Alur, R., Ivancic, F., Kim, J., Lee, I., Sokolsky, O.: Generating embedded software from hierarchical hybrid models. In: Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems, pp. 171–182. ACM Press (2003)
44. Alur, R., Grosu, R., Lee, I., Sokolsky, O.: Compositional Refinement of Hierarchical Hybrid Systems. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 33–48. Springer, Heidelberg (2001)
45. Giese, H., Henkler, S.: A survey of approaches for the visual model-driven development of next generation software-intensive systems. Journal of Visual Languages and Computing 17, 528–550 (2006)
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 Press, New York (2006)
47. Güdemann, M., Ortmeier, F., Reif, W.: Formal Modeling and Verification of Systems with Self-x Properties. In: Yang, L.T., Jin, H., Ma, J., Ungerer, T. (eds.) ATC 2006. LNCS, vol. 4158, pp. 38–47. Springer, Heidelberg (2006)
48. Goldsby, H.J., Cheng, B., Zhang, J.: AMOEBA-RT: Run-Time Verification of Adaptive Software. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 212–224. Springer, Heidelberg (2007)

Model-Based Reasoning for Self-Adaptive Systems – Theory and Practice

Gerald Steinbauer and Franz Wotawa[⋆]

Technische Universität Graz, Institute for Software Technology,
Inffeldgasse 16b/2, A-8010 Graz, Austria
{gstein,wotawa}@ist.tugraz.at,
http://www.ist.tugraz.at/

Abstract. Internal faults but also external events, or misinterpretations of sensor inputs as well as failing actuator actions make developing dependable systems a demanding task. This holds especially in the case where systems heavily interact with their environment. Even in case that the most common faults can be handled, it is very unlikely to capture all possible faults or interaction patterns at development time. As a consequence self-adaptive systems that respond to certain unexpected actions and observations at runtime are required. A pre-requisite for such system behavior is that the system itself has knowledge about itself and its objectives, which can be used for adapting its behavior autonomously. In order to provide a methodology for such systems we propose the use of model-based reasoning as foundation for adaptive systems. Besides lying out the basic principles, which allow for assurance of correctness and completeness of the reasoning results with respect to the underlying system model, we show how these techniques can be used to build self-adaptive mobile robots. In particular the proposed methodology relies on model-based diagnosis in combination with planning. We also discuss modeling issues and show how modeling paradigms influences the outcome of model-based reasoning. Moreover, we introduce some case studies of self-adaptive systems that rely on model-based reasoning concepts in order to show their applicability in practice. The case studies include mobile robots that react on hardware and software failures by applying corrective actions like restarting subsystems or reconfiguration of system parameters.

1 Introduction

What makes humans successful in interacting with other humans and nature and allows for reacting to situations, which have not been faced before? There are many potential answers like the ability to learn, to make associations, and to reason using the available knowledge that has been gained during a lifespan. Especially, reacting appropriately to new conditions is important and allows for exploring new territories. Such a capability would also be strongly desired for autonomous systems like mobile robots and are necessary if we want robots to be of more general use for us. A household robot for example, has to adapt to new homes and changes in the environment smoothly. This kind of self-adaption is not the only one we are interested in.

[⋆] Authors are listed in alphabetical order.

Consider for example a robot where a fault in the hardware occurs. For example, there is a problem with a connector causing the software to crash. Even in case the software is automatically restarted, the fault in the connector again leads to a crash. Without having control on possible repair actions allowing us to overcome such difficulties the robot would be lost. A similar situation would happen when a sensor delivers wrong values or an actuator that does not work as expected but the control system is not capable to deal with such faults and their consequences. It is worth noting that handling all possible faults in all situations explicitly via introducing hardcoded measures seems to be impossible or at least a very tedious task during design. Hence, again there is a strong requirement for self-adaption in the field of autonomous systems.

The main objective of the work described in the paper is to make an autonomous mobile system more robust and flexible. Therefore, all sources of malfunctioning, i.e., faults in the hardware or software, not considered interactions with the environment at design time, and wrong beliefs (which might originate from sensor data interpretations) have to be handled. Flexibility can be gained by allowing a system to continue its mission even in case of faults if they for example belong to system parts that are not important to fulfill a give mission. Alternatively, the system reconfigures itself in order to still provide the necessary functionality.

In order to achieve all of the mentioned objectives for enabling real autonomy the robot control system has to have means for anomaly detection, root cause analysis, and repair, which can be summarized as diagnosis. In anomaly detection the expected behavior in a current state is compared with the observed behavior. In model-based reasoning the expected behavior we obtain from a formal description of the system and its environment, i.e., the model. We assume that the models for behaviors are correct, i.e., they reflect the real actual behavior of the robot system. The observations in the robotics domain come from the sensors. Root cause detection is the part of diagnosis where the reason for a deviation between the expected and the observed behavior is identified. We make use of the fault detection model for root cause analysis. The last step of diagnosis is repair where appropriate measures for eliminating the undesired effects of a root cause are taken. It is worth mentioning that repair does not necessarily mean to replace a faulty component with a spare part. Repair can also be to take compensatory actions for eliminating undesired effects or reconfigurations of a system for enabling the wanted functionality.

So why is diagnosis of autonomous systems so difficult? The main reason lies in the increase of system influences that cannot be foreseen anymore. Almost all nowadays systems are designed for a particular purpose taking care of the design constraints, which are more or less limited by the purpose and the field of application. Most of the constraints are known and careful engineering allows for ensuring functionality as expected over lifetime. Of course maintenance work is necessary but due to knowledge about the system and its comprising parts even maintenance actives can be planned and managed. The situation changes completely when increasing autonomy or the field of application. In both cases the number of design constraints increases as well. As a consequence there is a high likelihood that interactions between the system and its environment, which are not considered during engineering, take place. Such interactions potentially lead to undesired behavior or even a failure.

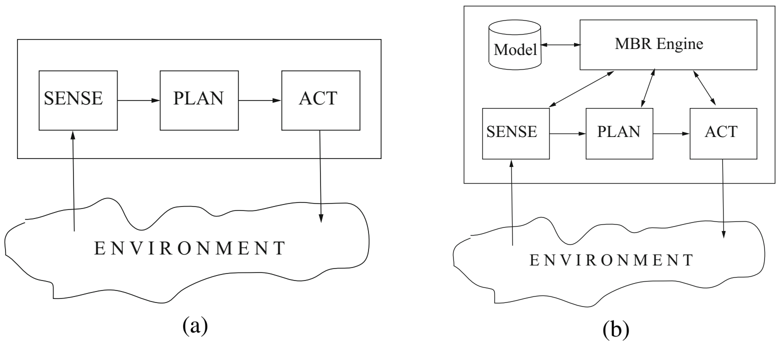


Fig. 1. The sense-plan-act control architecture (a) and its adaptive variant (b)

When it is not possible to keep all necessary design constraints in mind, we have at least to think about alternatives. In this paper we suggest to move some of the hard coded decisions a system can take to a more flexible control system that relies on a model of the system as well as its environment. We argue that unforeseen interactions with the model-enhanced system might have not the same negative impacts. The reason lies in the fact that the model describes the functionality, the behavior, and other prerequisites in a declarative way. The model does not describe how to solve a certain problem. Hence, in case of a misbehavior possible solutions can be derived from the model directly using a declarative reasoning engine. Of course the reactions to events that are not considered might be limited if the models do not reflect the behavior of the overall system.

In this paper we discuss model-based reasoning and its application to self-adaptive systems in the context of autonomous mobile robots. We extend the standard sense-plan-act control paradigm depicted in Figure 1(a) with a model-based reasoning engine (Fig. 1(b)). In the original control paradigm actions are selected on bases of the current state, which is determined by the measurements obtained from the sensor and the internal belief, and the mission (or goal) of the robot. The action selection is done by the planning module and the action module executes the actions. As a consequence of action execution the environment is changed, e.g., by driving from a position to another, which is again measured using the sensors. Hence, there is always a feedback between action execution and sensing actions in robot control. Although, there are many other control paradigms for robots available, e.g., Rodney Brooks's subsumption architecture [1], the underlying principle of actions selected because of sensor information and internal belief remains the same.

Given the robot's control loop we are able to identify three kind of faults that can arise during a mission. First, the sensors might deliver wrong values. Wrong in this context mean that they do not reflect the state of the environment. For example, a robot playing soccer sees the ball at a position but the ball is not there. Second, the execution of actions might fail but remains undetected. An example for this fault is that a robot want to release and object but fails and still carries the object with her. These two kind of faults have as consequence a faulty internal belief, which might lead to wrong plans and dreadful control decisions causing a mission to fail. The third fault category that is relevant are faults in the hardware or software. Note that we do not require a design error causing later on a fault and a failing behavior of the robot. Due to interactions with the environment, e.g., when being hit by another entity, the robot hardware might be affected.

The extended control paradigm given in Fig. 1(b) adds a model-based reasoning (MBR) engine. The MBR engine makes use of a model of the system in order to correct the execution in case of a fault. The model is assumed to capture the structure of the system and the behavior of the components. Moreover, system properties as well as environmental constraints should be part of the model in order to detect situations that are impossible in reality. The same model is used for detecting inconsistencies as well as for fault localization and repair. For the latter the knowledge about functionality of system parts, the mission, and repair actions have to be added. In the following section we introduce model-based reasoning and how diagnosis can be integrated into the planning process.

There are many challenges to be solved when using MBR for the purpose of self-adaptive systems. First of all, developing a model is a hard task. The model should be compositional, declarative, and capture the function and behavior of the system in a depth allowing to use an MBR engine for fault detection, localization, and repair. However, if the model is implemented using basic design principles like the *no function in structure* principle, and *compositionally* large parts of the model can be re-used in other systems. The second challenge is due to underlying reasoning mechanism, which is based on non-monotonic reasoning. The used reasoning technique allows for stating hypotheses about the health state of components. Such hypotheses are asserted and retracted for computing root causes. The computation complexity of underlying algorithm is high in the worst case, and counter measures like the use of heuristics or certain optimality criteria have to be undertaken. Note that in practice diagnosis of systems comprising hundreds to several thousands of components is possible in a reasonable time. Using integrated circuit examples from industry (ISCAS benchmark suite) it had be shown that diagnosis of single faults in circuits with up to 1000 components can be calculated in less than 200 milliseconds on a standard computer while diagnosis of triple faults can run for up to 5 minutes [2]. If real-time requirements have to be considered, e.g., react on signals within a few milliseconds, special domain-dependent encoding of the diagnosis problem can be used [3].

In summary we provide the following contributions:

- We discuss an extension to the well-known sense-control-act architecture used in autonomous systems. In particular, we add an explicit component for handling faults, external events, and misinterpretation of sensor inputs.
- We formally introduce the underlying ideas and basic concepts of model-based reasoning and illustrate them using an example from the robotics domain.
- Furthermore, we introduce an algorithm for diagnosis and self-adaptation. The self-adaptation algorithm combines diagnosis and planning in order to implement the extended sense-plan-act control architecture.

The rest of this paper is organized as follows. In the next section we discuss research related to MBR and self-adaptive systems. We continue with a formal introduction to

MBR. There we explain the underlying definitions using a running example. Moreover, we state a simple algorithm for fault localization and show how this algorithm can be combined with the planning component of the robot's control architecture. In order to show the applicability of MBR to self-adaptive systems, we recall several applications. In particular we show how software can be diagnosed using MBR and how to perform fault localization for particular robot drives. Finally, we conclude the paper.

2 Related Research

The goal to make an autonomous system comprising hardware or software dependable in the sense that the system is able to react to neither modeled nor foreseen circumstances has a long tradition in the areas of Artificial Intelligence, Robotics and Software Engineering.

In Robotics the question how decision making and execution can be robustly organized despite poorly modeled tasks and environments, unreliable perception, and non-deterministic execution arose early. The three-layer-architecture proposed in [4] is basically still the foundation for most robot control architectures. There the authors introduced the idea to structure decision making and execution into three layers with increasing abstraction, deadlines and computational demands. The controller level reacts directly and fast on sensor inputs and is able to cope with local and immediate problems like sensor glitches. The sequencer level is responsible for triggering primitive behaviors in order to achieve a given task. The sequence of behaviors to be executed is generated by the deliberative level. Whereas the deliberate level is able to incorporate persistent changes of the environment or the robot's capabilities in its planning, the sequencer level is able to deal with problems in the outcome of behaviors by issuing a re-planning request to the deliberative level. This approach is basically similar to the proposed method introduced in this paper. However, it differs in two ways. In particular, using a reasoning engine more precise information on the root cause of a problem can be obtained that can be used by the sequencer and deliberative level to achieve more fine-grained modifications of planning and plan execution. For instance, some faulty actions may be excluded from the planning process. Moreover, the reasoning engine is able to deliver information on the proper function of system components, which can be considered by the deliberative level or for reconfiguring the sequencer and controller level. We will present an example for reconfiguration of defective robot hardware later. Such interactions with the system are nor foreseen in the original architecture.

Using the ideas of the three-layer-architecture Kramer and Magee [5] presented a proposal for an architecture for self-adaptive systems. This proposal comprises layers for component control, change management and goal management. The component control is responsible for the reconfiguration of individual components. Such reconfiguration requests are initiated by change management in order to react to states reported by the layer below or to new goals. Action at this layer comprises the creation or termination of components. The goal management is intended to produce plans for the change management layer in order to achieve certain goal or requirements for the overall software system. The main difference of this architecture to the approach presented in the paper is that it deals mainly with the structure of the system. For instance the architecture may take care that a minimum number of redundant modules run at the same time. The architecture does not determine how the software system fulfill its mission. In the approach presented in this paper the planning for achieving a mission and the reaction to problems in the system and its environment are interwoven in one decision making and execution module. Therefore, both parts can benefit from each other. For instance, the mission planner can use information about the current functionality of a hardware component.

In [6] Avizienis and colleagues provided ed definitions for the taxonomy of faults, their sources, their properties, and techniques to cope with them. The taxonomy origins from the secure computing domain and distinguishes basically two type of faults: development and operational faults. The former faults are maybe maliciously introduced in the design and implementation phase whereas the latter faults arise during the operation of the system. The faults mentioned in the introduction the proposed approach is able to cope with (sensing, execution, hardware and software) belong to the former group. We consider development faults only to the extent that they lead to transient problems and can be handled for instance by repeating an action. Even if the taxonomy of [6] considers interaction it is limited to direct interaction of components. In the context of autonomous system a more important source of faults is the interaction with the environment. Moreover, faults in the autonomous decision making system are not considered and posts several new challenges. In [7] an extended taxonomy of faults suitable for autonomous systems was presented. In [8] the taxonomy was enhanced by faults that origin from properties of the used algorithms. Many of the algorithms used today in Robotics are probabilistic methods. Therefore, there is a chance that even the algorithm is implemented correctly it produced inferior results. Moreover, the paper reports a survey of the occurrence of faults and their impacts in real robot systems.

There are many other applications of MBR for adaptive systems. All of the work we are discussing in this section deals with MBR at runtime or at least can be applied to be used at runtime. The first papers deal with on-board diagnosis of cars. Cascio and colleagues [9] as well as Struss and colleagues [10–13] developed systems that are able to find and locate faults of car subsystems during operation. Breaking systems as well as the whole combustion system of cars were modeled. Due to limited computational resources and real-time requirements the authors rely on simplified models that capture the essential parts of the car subsystems but ignoring the real quantitative values. In particular qualitative models [14] are used, where the quantitative values are mapped to qualitative values. For example, instead of considering temperature values like -4 degrees Celsius a qualitative representation like "*cold*" is used. In their papers the authors demonstrated that the combination of qualitative reasoning models and MBR can be effectively used for on-board fault detection and localization in cars. The repair task, which would follow fault localization, was not handled.

As part of NASA's Deep Space 1 mission a control system for spacecrafts that has capabilities to adapt itself to faults has been introduced [15, 16]. The underlying idea is to gain autonomy for spacecrafts in order to make increase applicability. The control system comprises a MBR engine that allows for fault localization and re-configuration [17] and a reactive planner [18]. The whole system was implemented and successfully tested during the Deep Space 1 mission. The remote agent experiment is a proof of concept that MBR technology is capable of increasing autonomy and a good bases for self-adaptive systems.

Another domain that is closely related to self-adaptation is reconfiguration where parts of a system are changed to implement a given functionality of the system. This might be changes in the structure of the system or changes in the parameters such that the functionality of the system is adapted to given needs. Configuration and reconfiguration is a still very active research field. Early work that also deals with MBR includes Crow and Rushby [19], and Stumptner and Wotawa [20]. The latter one deals with parameter reconfiguration and uses different component modes to determine valid parameters for the desired functionality. In the paper the authors introduce the foundations for dynamic adaptations of parameters in case of changes in the functionality.

The WS-Diamond project[1] deals with diagnosis and repair of web-services, workflows, and service-oriented architectures. The objective of the project was to achieve self-healing capabilities of web-services [21] where repair is not only a replacement of software components or services but also requires compensating actions for bringing the faulty system again into a correct state. Moreover, the project also focusses on general questions like diagnosability and repairability when using certain models of the system [22]. There are many similarities between WS-Diamond project and our work in self-adaption of autonomous systems. There is a need for combining diagnosis and planning. Moreover, some of the models used for diagnosis of web-services can also be used for software diagnosis of a robot's control program. However, there are also differences that come from the application domain. The environment where web-services are running is more restricted than the real world. In addition faults coming from hardware or the interaction between faulty hardware and software have no counterpart on side of web-services.

Karsai and colleagues [23] introduced a approach for integrating diagnosis and control. Their work is based on Bond graphs for modeling systems in the aeronautic domain. Their work is also based on the underlying ideas of MBD. In [24] the authors present the basic principles using a case study from a fuel system of an airplane. In the domain of model driven development and advanced mechatronic systems the authors of [25] introduced an approach that allows to combine multiple modeling methods to model complex reconfigurable hybrid systems.

3 Model-Based Reasoning

The Artificial Intelligence (AI) field Model-based reasoning (MBR) deals with foundations for obtaining solutions directly from the available knowledge of the real world. The available knowledge, i.e., the model, represents the structure and the behavior of those parts of the world that are necessary to know in the context of the application. The restriction to the context of application is not really a problem because all models like the theory of physics only represent the parts of interest. Using a model directly is in contrast to very early work in AI where knowledge about how to obtain a solution from given facts is used. Hence, in MBR the model does not capture the way of how to

[1] see `http://wsdiamond.di.unito.it/`

obtain solution but states constraints between the model entities. Because of the direct use of model MBR is also called reasoning from first principles.

Using a model directly for obtaining solutions increases flexibility, which is necessary in certain application domains like adaptive systems where the degree of adaption as consequence of events might not be known in advance. It is worth noting that there was a strong need for a flexible method because traditional AI techniques like rule-based systems cause high costs in case of adaptions as part of the usual maintenance process on the available knowledge base. In the early 80th of the last century the foundations of MBR were first published [26–28]. In most of these early papers diagnosis is the underlying application domain. In this paper, we also make use of model-based diagnosis (MBD) and rely on the foundations that goes back to Reiter's [29], and De Kleer and William's [30] work.

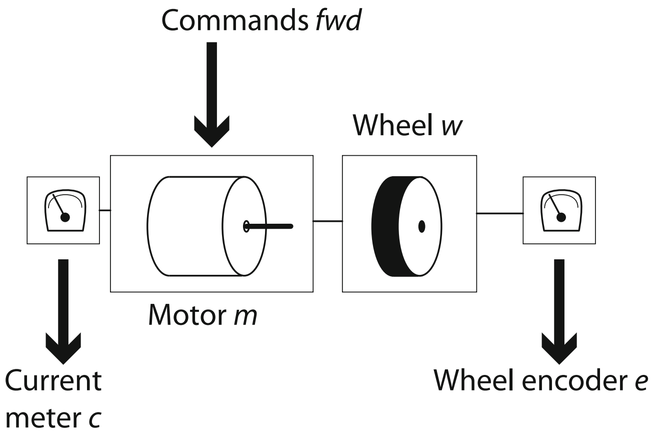


Fig. 2. A partial schematics of a robot drive

Before introducing the formal definitions and an algorithm for MBD, we discuss the underlying idea using a small example from the Robotics domain taken from [31]. Figure 2 depicts the schematics of a part of a robot drive comprising a current meter c, a motor m, a wheel w, and a wheel encoder e. The current meter c is for measuring the current flowing through the motor m. In case of setting m to start moving in a forward direction using the command fwd, there will be a current. Otherwise, we do not measure any current using c. If the motor is running, the wheel w should rotate, which can be measured using the wheel encoder e. Now let us assume that we start the motor using the fwd command, and observe a nominal current flow using c but no information coming from the wheel encoder e indicating a rotating wheel.

Using the given information we are able to identify diagnosis candidates. Because of a measured current flow we can eliminate the cause motor broken when assuming single faults only. A single fault means that there is only on root cause for on or more undesired observations. For instance, if the battery of the robot is faulty several components might show an undesired behavior even the components are correct. Hence, only the wheel or the wheel encoder might be faulty. Of course it can also be the case that an assembly between the wheel or the motor or the encoder might be the reason.

However, in our model we have not introduced the assembly as an own entity. What we can conclude from the example is the following: (1) A diagnosis candidate is a component that when being faulty explains the given observations; (2) We used the behavior of the components and their connections for identifying diagnoses; (3) It is possible to exclude potential causes like the motor from the list of diagnosis candidates; (4) For the purpose of reasoning we used the underlying assumption that a component is correct or that it is faulty. For example, when assuming the wheel w to be broken, there will be no rotation and thus no information coming from the wheel encoder e even in case e is working as expected.

We now formalize the basic idea of MBD. We start with the definition of diagnosis models, which are also called system descriptions (SD) in the MBD context. A diagnosis model comprises the structure of the system and the behavior of components in case the component works as expected and alternatively in case the component is in a particular faulty state. For the case of correctness we use the negation of the predicate AB where AB stands for abnormal. It is worth noting that in case of abnormal behavior, i.e., when AB is true, we do not know the exact behavior. Hence, there are no rules in SD for this case. Alternatively, a component is in a known fault state (or fault mode) F_i. For this purpose there will be a predicate F_i and a set of rules specifying the faulty behavior. For each component $c \in COMP$ we have a set of possible modes where each mode represent a particular state. AB and $\neg AB$ is always part of the possible states. In order to access these states we assume a set $MODES$ that comprises all possible states, and a function $modes : COMP \mapsto 2^{MODES}$ mapping components to the possible modes. For $modes$ we require that $\forall c \in COMP : \{AB, \neg AB\} \subseteq modes(c)$ holds.

In the following we introduce the model using first order logic for our example. Note that the approach is not restricted to first order logic. Any logic that allows for proving satisfiability works.

Motor: In case of a correct behavior a particular motor is running in forward direction if there is command for moving forward cmd_fwd. In this case there is a current flowing through the motor. This current can be of nominal value or higher if there is a strong resistance coming from attached components like wheels. We assume an abstraction function from the quantitative measurement signal to the qualitative observation. Usually continuous signals are noisy and dynamic. Therefore, we use a hysteresis-based approach for the abstraction step [32]. Hence, for all components X that are motors, the following logical sentence formalize their behavior:

$$\forall X : motor(X) \rightarrow \left(\begin{array}{c} \neg AB(X) \rightarrow (cmd\_fwd \rightarrow direction(X, fwd)) \\ direction(X, fwd) \rightarrow torque(X, fwd) \\ direction(X, fwd) \wedge resistance(X) \rightarrow current(X, high) \\ direction(X, fwd) \wedge \neg resistance(X) \rightarrow current(X, nominal) \\ current(X, high) \vee current(X, nominal) \rightarrow direction(X, fwd) \end{array} \right) \quad (1)$$

Wheel: The torque provided via the axle is turned into rotations of the wheel. Only in case of a stuck wheel there is a resistance against the applied torque. But this case is not going to be formalized because we are only interested in the correct behavior. For wheels X we formalize the correct behavior as follows:

$$\begin{array}{l}\forall X : wheel(X) \rightarrow \\ (\neg AB(X) \rightarrow (torque(X, fwd) \rightarrow (rotate(X, fwd) \wedge \neg resistance(X))))\end{array} \tag{2}$$

Current meter: The current meter measures the value of the current. Hence, there is a one to one correspondence of the current flow and the measurement in case of a correct behavior. This behavior of a current meter X can be formalized as follows:

$$\begin{array}{l}\forall X : current\_meter(X) \rightarrow \\ \left(\begin{array}{c}\neg AB(X) \rightarrow (current(X, high) \leftrightarrow observed\_current(X, high)) \\ \neg AB(X) \rightarrow (current(X, nominal) \leftrightarrow observed\_current(X, nominal))\end{array}\right)\end{array} \tag{3}$$

Wheel encoder: The wheel encoder X is providing a frequency only in case of a rotation. For a wheel encoder X we introduce the following logical sentence:

$$\begin{array}{l}\forall X : encoder(X) \rightarrow \\ (\neg AB(X) \rightarrow (rotate(X, fwd) \leftrightarrow frequency(X)))\end{array} \tag{4}$$

What is now missing to complete the model for our example is a description of the structure. We first define the components:

$$motor(m) \wedge wheel(w) \wedge current\_meter(c) \wedge encoder(e) \tag{5}$$

Second, we have to define the connections between the components. This can be done using the following logical sentences:

$$\begin{array}{r}torque(m, fwd) \leftrightarrow torque(w, fwd) \\ resistance(m) \leftrightarrow resistance(w) \\ current(m, nominal) \leftrightarrow current(c, nominal) \\ current(m, high) \leftrightarrow current(c, high) \\ rotate(w, fwd) \leftrightarrow rotate(e, fwd)\end{array} \tag{6}$$

The rules stated in Equations (1) – (6) define the structure and behavior of the model SD of our small example. For this example the set of components comprise 4 elements, i.e., $COMP = \{m, c, w, e\}$. Moreover, the example makes only use of AB and $\neg AB$. Hence, the set of modes only holds these elements: $MODES = \{AB, \neg AB\}$.

We now define a diagnosis problem formally. Although, we borrow the ideas behind model-based diagnosis from Reiter [29], we adapt the definitions in order to deal with fault states. In Reiter's seminal work only the correct behavior of components is used. There are many papers also dealing with fault modes like [33] and [34]. Both papers deal with the relationship between the case of diagnosis without fault modes and the one with only explicit fault modes, i.e., where there is no explicitly given fault behavior. Fault modes usually improve diagnosis, i.e., they help to reduce the number of diagnosis candidates. It is worth noting that sometimes the use of fault modes is not necessary for improving diagnosis. Handling physical impossibilities seems to be sufficient. See [35] for a more detailed discussion on this topic.

Definition 1 (Diagnosis problem). *A diagnosis problem is a tuple* $(SD, COMP, MODES, modes, OBS)$*, where* SD *is a model,* $COMP$ *a set of components,* $MODES$ *a set of modes, and* $modes$ *a function mapping components to their modes, and* OBS *a set of observations.*

The diagnosis problem for the running example from Fig. 2 comprise the system description, the component set, the modes and the $modes$ function, which map each component to the set $\{AB, \neg AB\}$. The observations for this example are: $\{cmd\_fwd, observed\_current(c, nominal), \neg frequency(e)\}$.

A diagnosis in contrast to the original definition of Reiter [29] is a mapping of exactly one mode to each component such that the observations can be explained. We first define the term mode assignment.

Definition 2 (Mode assignment). *Given a diagnosis problem* $(SD, COMP, MODES, modes)$*. A mode assignment* Δ *is a set where the following properties hold:*

1. $\forall c \in C : ((\exists m(c) \in \Delta) \wedge m \in modes(c))$
2. $|\Delta| = |COMP|$

Mode assignments are used to specify that a component is in a certain state. Note that for simplicity we do not consider temporal changes in mode assignments. If temporal changes of behavioral states are possible, i.e., in case of non-permanent faults, the definitions have to be extended.

We are now able to define a diagnosis using mode assignments. The underlying idea here is to find a certain state of the system, which comprises the states of each component, such that the corresponding behavior is not in contradiction with the given observation.

Definition 3 (Diagnosis). *Given a diagnosis problem* $(SD, COMP, OBS)$*. A mode assignment* Δ *is a diagnosis if and only if* $SD \cup OBS \cup \Delta$ *is satisfiable.*

For our running example the mode assignment $\{\neg AB(m), \neg AB(c), AB(w), \neg AB(e)\}$ as well as $\{\neg AB(m), \neg AB(c), \neg AB(w), AB(e)\}$ are both a diagnosis.

In practice computing all diagnoses is not required. Instead we are interested in the "most likely" diagnoses or the diagnoses comprising the least faults. In order to define such ranking of diagnoses we make use of some auxiliary definitions. We first define a set $S_m(\Delta)$ for a mode assignment Δ where $m \in MODES$. The set S_m comprises all components in the mode assignment with mode m, i.e., $S_m(\Delta) \equiv_{DEF} \{c | m(c) \in \Delta\}$. We also assume the existence of a function $p_m : COMP \mapsto [0, 1]$ that maps components to its probability of being in state $m \in MODES$. Obviously $\sum_{m \in modes(C)} p_m(C) = 1$ must hold for all components $c \in COMP$ in order to reflect that a component c has to be in exactly one state.

For the definition of minimality with respect to the set S_m of components of a diagnosis that are in mode m we make use of the following thought. A diagnosis should be preferred over another diagnosis if the first one assumes less faulty components. In our case faulty components in a diagnosis have assigned a state m that is not $\neg AB$. There are two possibilities to state minimality using this thought. We first define subset minimality which ensures that there is no diagnosis comprising less correct diagnoses than the given diagnosis.

Definition 4 (Subset minimality). *A diagnosis Δ for a given diagnosis problem is subset minimal if and only if there exists no other diagnosis Δ' for which the following proposition hold: $S_{\neg AB}(\Delta') \supset S_{\neg AB}(\Delta)$.*

Alternatively to subset minimality we might think about preferring diagnoses that are the smallest ones. This definition of minimality is based on the cardinality of sets.

Definition 5 (Minimal cardinality). *A diagnosis Δ for a given diagnosis problem is minimal with respect to cardinality if and only if there exists no other diagnosis Δ' for which the following proposition hold: $|S_{\neg AB}(\Delta')| > |S_{\neg AB}(\Delta)|$*

Minimality with respect to cardinality can be also seen as a global minimality, because there are no diagnoses with less faulty components.

The last definition of minimality is based on the probability of a diagnosis. When assuming that a mode of a component is stochastically independent from each mode of another component, then the probability of a diagnosis Δ is nothing else than the product of the probabilities of all single mode assignments in Δ, i.e., $Prob(x) \equiv_{DEF} \prod_{m(c)\in\Delta} p_m(c)$. The independence assumption is quite reasonable even if component interacts as the probability express the chance that a component fails by itself.

Definition 6 (Most likely diagnosis). *A diagnosis Δ for a given diagnosis problem is the most likely diagnosis if and only if there exists no other diagnosis Δ' for which the following proposition hold: $Prob(\Delta') < Prob(\Delta)$*

All the above minimality definitions are of practical use. In the hardware domain where fault probabilities of components are known a most likely or most probable diagnosis might be the right choice. In situations where the probabilities are not known, minimality with respect to the size of faulty components might be more appropriate. It is also worth noting that probabilities can also be used in the context of model-based diagnosis to determine the next measurements required to distinguish diagnoses. We refer the interested reader to Williams and de Kleer's paper [30] for an introduction to a method for optimal measurement selection with respect to the required number of measurements for locating the fault.

In the following we introduce an algorithm that computes minimal diagnoses up to a predefined number of faulty components. The algorithm starts assuming that all components are working as expected. In case of a contradiction with the given observations, one after the other component is assumed to switch its mode. Again for each of these mode assignments a check of consistency is performed, and so on. The process stops either when the bound is reached or when there are no combinations left. It is also worth noting, that a check is performed whether there exists a small diagnosis where the correct components are a superset of the correct components of the currently computed diagnosis. If this is the case the new diagnosis is removed from the list of candidates, thus ensuring subset minimality.

In Figure 3 we introduce the MBDIAG algorithm. The algorithm assumes that we have a theorem prover that allows for checking consistency. This is done by calling the CHECK method. It is worth noting that the term theorem prover is not only for representing a system for checking satisfiability of a logical formulae. Instead any algorithm

```
Algorithm MBDIAG
Input: A model $(SD, COMP, MODES, modes)$, observations $OBS$, and the number $n_{AB} \geq$
0 of faulty component to be searched for.
Output: A set of diagnoses $R$.

1. Let $R$ be the empty set.
2. Let $DS$ be the set comprising the element $\{\neg AB(c) | c \in COMP\}$, and let $n = 0$.
3. Let $DS'$ be the empty set.
4. For all elements $\Delta \in DS$ do:
   (a) If CHECK$(SD, \Delta, OBS)$ is consistent, then add $\Delta$ to $R$ if there exists no subset minimal $\Delta'$ in $R$.
   (b) Otherwise, for all components $c$ where $\neg AB(c) \in \Delta$ do:
       i. For all $m \in modes(c) \setminus \{\neg AB\}$ add $\Delta'$ to $DS'$ with $\Delta'$ is $\Delta$ where $\neg AB(c)$ is changed to $m(c)$.
5. Let $n = n + 1$ and let $DS$ be $DS'$.
6. If $n \leq n_{AB}$ and $DS \neq \emptyset$, go to 3
7. Otherwise, return $R$ as result.
```

Fig. 3. The model-based diagnosis algorithm MBDIAG

that allows for deciding whether a given system state together with a formalized model is contradicting the given observations or not. Hence, a simulator or a constraint solver can also be used for this purpose.

Obviously MBDIAG terminates because either the bound n_{AB} is reached or no new diagnosis candidates have to be checked. Moreover, MBDIAG has to compute subset minimal diagnoses because the others are not added to the result set R. The worst case complexity is of course exponential in the number of components. However, for single faults the algorithm is feasible requiring $O(|COMP| \cdot |MODES|)$ steps. Even in case of double and triple faults MBDIAG is in the worst case polynomial in the number of components and modes.

There have been many other diagnosis algorithms described in literature. The most closely to ours is Reiter's diagnosis algorithm [29] that was corrected by Greiner et al. [36]. The algorithm is based on conflicts and hitting sets. A conflict basically is a set of components that when assuming to behave correctly reveal an inconsistency. By combining all these conflicts (via the computation of hitting sets) all minimal diagnoses can be obtained. Another approach that is based on an assumption based truth maintenance system [37] was introduced by de Kleer and Williams [30]. Other papers introducing algorithms for model-based diagnosis include [38], [39], [40], and [41]. The last three publications are of particular interest because they make use of constraints for representing models and observations.

3.1 Smart Control

What is missing after performing fault detection and localization is repair. There are many ways to bring a system to a correct state after a failure. The most straightforward way is replacing broken components with their spare parts. This cannot be done in all application domains. Consider for example a spacecraft on mission. Replacing a component there is obviously not a feasible option. Repair can also be performed via changing the system's structure or configuration in order to meet the current needs. In this case broken components are excluded and other components take their place. In practice usually for this case redundant components are used. However, it is also often possible to use implicit redundancy within an application to make a replacement that allows for still reaching mission goals. For example, if communication via WiFi is not possible any more, maybe it is possible to communicate via another means for communication, which might not be available always but at least sometimes. Hence, performance of the overall system will degrade but not to a point where the mission is in danger.

For coming up with a smart repair engine knowledge of possible repair actions as well as the provided functionality and the goal has to be formalized. Moreover, the results of diagnosis have also to be considered. We do not discuss planning and the planning problem in detail here and refer the interested reader to other publications [42, 43]. Instead we discuss smart plan execution, which is required for really smart control. Combining MBD and planning is not new. Sun and Weld [44] introduced the use of planning for diagnosis with the purpose of controlling the diagnostic process itself. Friedrich and Nejdl [45–47] formalized the process of diagnosis and repair, and also distinguishes (repair) actions from observations. Our work is based on the previous work and focusses more on the execution of control using the sense-plan-act paradigm (see Fig. 1) of mobile autonomous robots.

In the following we introduce the algorithm EXT_PSA_ARCH [31] that is depicted in Figure 4. The algorithm makes use of a function SENSE, which returns the current state of the autonomous system comprising the internal state and the information obtained from the sensors. We assume that only reliable sensor information is given. Therefore, if a diagnosis indicates a sensor fault, the particular sensor is ignored and its information is not provided anymore. Although time is not handled explicitly in the algorithm it is worth noting that time exceeds during execution. Thus SENSE represents the state at a discrete point in time only, where measurements and the internal state are observed.

The control algorithm starts computing a plan based on the available information, i.e., the current state provided via the SENSE function, and the planning knowledge based M_p together with the goal state S_G. At the beginning all actions that can be performed by the system are functioning, and are therefore available for planning. Afterwards, the plan is executed by sequentially executing the actions. First, the pre-conditions of the current action a are checked. In case that the pre-conditions are not fulfilled in the current state, the action cannot be performed. This can happen due to an external event. As a consequence, re-planning has to be performed and plan execution starts again using the new plan.

If the pre-conditions of an action are fulfilled, the action is executed. This execution might be terminated returning a failure. In this case diagnosis has to be performed that returns diagnoses using the MBDIAG method, from which a leading diagnosis is obtained. A leading diagnosis can be either the smallest diagnosis or the one with the highest fault probability. For simplicity we do not handle the case of multiple diagnoses that cannot be distinguished with the available information. If such case occurs,

```
Algorithm EXT_PSA_ARCH
Input: A planning model $M_p$, a diagnosis model $(SD, COMP, MODES, modes)$, and the goal state $S_G$.
Output: Computes and executes a plan from the current state to $S_G$.

1. Let $p$ := PLAN($M_p$,SENSE(),$S_G$).
2. While $p$ is not empty do:
   (a) Let $a$ be the first action of plan $p$.
   (b) Remove $a$ from $p$.
   (c) If the pre conditions of $a$ are not fulfilled in SENSE(), then go to 1
   (d) Otherwise, execute $a$.
   (e) If the execution terminates with a failure, let $S_\Delta$ be the result of calling
       MBDIAG($SD, COMP, MODES, modes$, SENSE()), and $\Delta$ be the leading diagnosis of $S_\Delta$.
   (f) $\Delta$ indicates a sensor failure only, then consider $a$ to be executed without failure and
       proceed. Otherwise, remove all actions from the planning model that cannot be longer
       used because of diagnosis $\Delta$. Go to 1.
   (g) If the effects of $a$ are not fulfilled in SENSE() , then go to 1. Otherwise, continue
       executing the plan.
```

Fig. 4. The smart plan execution algorithm EXT_PSA_ARCH

measures for distinguishing diagnoses have to be performed, like providing testing procedures. For example, Wotawa et al. [48] introduce distinguishing test cases for solving such problems. Hence, we assume that we can always determine a leading diagnosis. This leading diagnosis is used to either remove actions that cannot be performed anymore, or to assume that some sensor data is no longer reliable. In the latter case there is no need to apply re-planning. Instead the action is forced to be executed and the procedure continues.

The last possibility for a fault occurring during execution is that the effects of an action are not visible. In this case we again perform re-planning, which might lead to the case where the current action is re-executed again. Note that this might lead to a situation where the robot is executing an action again and again due to a sensor failure or an external event. Therefore, in the implementation the repetition of executions of the same actions should be tracked and handled appropriately. For example, a diagnosis step might also be performed.

4 Example Applications

After introducing the foundations of MBR we discuss two of our applications of MBR in the domain of self-adaptive systems. The applications are in the field of autonomous mobile robots. Their common objective is to provide adaptation in case of faults occurring either in software or hardware. Although both applications share the same methodology, their underlying models are different. In software repair the dependencies between parameters that are passed from one software component to another are used to localize the fault. Whereas in the case of adaptive kinematics control the whole kinematics of a robot drive is modeled, which requires the use of difference or differential equations. Hence, the application also demonstrate very well the bandwidth of models that are used in MBR.

4.1 Software Repair at Runtime

We first discuss the application of MBD to software repair performed during runtime on a mobile robot. The work of Steinbauer et al. [49] we are discussing in this section distinguishes the fault detection from the fault localization part of diagnosis and is therefore different to ordinary MBD where both tasks are usually handled using the same model. The general idea behind the approach is to recover from severe failure as fast as possible. What the authors had in mind was to tackle cases like deadlocks or crashes of software components that would lead to a loss of the mobile robot.

Figure 5 shows the dependencies of the underlying robot control architecture taken from [49]. We see the different components and their potential interactions via messages and events they are sharing. In the following we discuss the approach used for software repair comprising fault detection, localization, and the repair step, where fault detection starts fault localization, which itself starts repair afterwards.

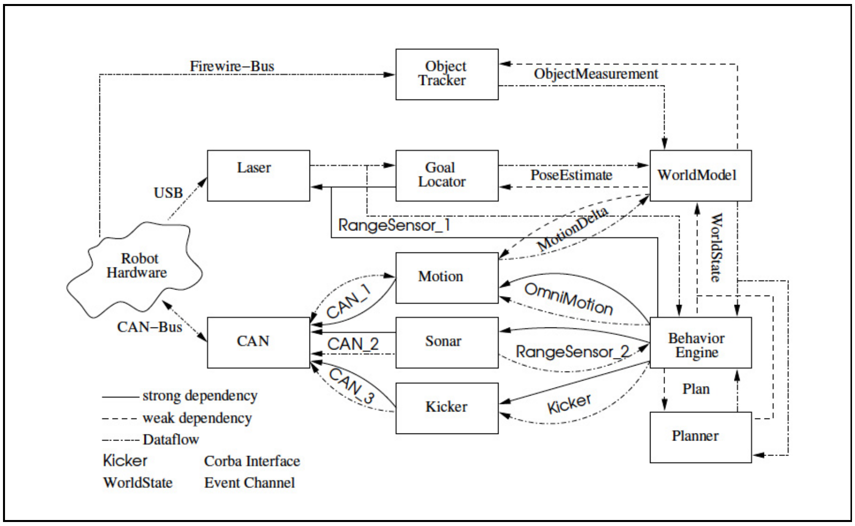


Fig. 5. The robot control architecture from [49]

Fault detection: In [49] fault detection is handled using monitoring. In particular a watch-dog process is invoked, which takes care of messages that are exchanged between the components of the control software, and processes and threads necessary for execution. The watch-dog process checks the following information during runtime:

– Periodic event production, e.g., the motion service has to produce an event every 50 ms.

- Conditional event production, e.g., there has to be an event WorldState produced by the world model component after an event ObjectMeasurement occurs.
- Periodic method calls, e.g., the sonar service of the robot should be regularly called by the behavior engine.
- Spawn processes, e.g., the motion service invokes exactly 6 threads.

The watch-dog raises the fault localization method in case a check fails. In addition every check that fails also has corresponding observations that are given to the fault localization method. The observations state the correctness or incorrectness of certain messages. For example, if the range sensor data is checked to be faulty, a $\neg ok(RangeSensor\_2)$ observation is generated. It is worth mentioning that the overhead for running the monitoring process can be neglected. In [49] the authors reported less than 1$\%$ overhead of CPU time and less than 5 $\%$ of memory consumption.

Fault localization: The fault localization step takes the outcome of detection as input and tries to identify the root cause. Because of runtime requirements and the fact that not the whole behavior of the software system (without replicating the whole program in logic) is available, a simplified model was used in [49]. The model itself only takes care of dependence information. A component has a dependence relation with another component if there is a message flow. The underlying idea is not new in the software domain. Friedrich et al. [50] used data and control dependences for locating bugs in VHDL programs. Later Wotawa [51] proved the equivalence of Mark Weiser's slicing [52] and the dependency-based model.

The idea of the dependence model is as follows. Every component that is correct, should produce a correct output in case the inputs are correct too. We only need to formalize this idea where *ok* ($\neg$*ok*) is used to represent a correct (incorrect) value of either an input or output. We describe the model behind the fault localization step using a small example. Consider the software architecture from Figure 5 and focus on the components CAN, Motion, and Sonar only. Between CAN and Motion messages are send on connection CAN_1 and between CAN and Sonar messages are send on connection CAN_2. Since we are only interested in software faults, no information of the hardware is given, and therefore the CAN component has no input in our model. Hence, we finally obtain the following system description SD for the small subsystem:

$$\begin{array}{l} \neg AB(CAN) \rightarrow ok(CAN\_1) \\ \neg AB(CAN) \rightarrow ok(CAN\_2) \\ \neg AB(Motion) \wedge ok(CAN\_1) \rightarrow ok(MotionDelta) \\ \neg AB(Sonar) \wedge ok(CAN\_2) \rightarrow ok(RangeSensor\_2) \end{array}$$

In the model we have three components $COMP = \{CAN, Motion, Sonar\}$, each having assigned two modes AB and $\neg AB$. Let us now assume that the fault detector checks fail for the signals MotionDelta and RangeSensor_2. In this case we have the following set of observations:

$$OBS = \{\neg ok(MotionDelta), \neg ok(RangeSensor\_2)\}$$

Obviously when assuming all components to be correct, we obtain a contradiction. It is easy to show that only the following two mode assignments are subset minimal diagnoses:

$$\{AB(CAN), \neg AB(Motion), \neg AB(Sonar)\}$$
$$\{\neg AB(CAN), AB(Motion), AB(Sonar)\}$$

When considering the smallest root cause first, only the diagnosis indicating that the CAN module is faulty remains.

Repair: After localizing the fault Steinbauer et al. introduced a repair approach where the restart faulty components. In some cases it is also necessary to restart components that are connected with the faulty ones. The information whether this is necessary is stored in the strong dependencies (see Fig. 5). If a component like Can fails also the Sonar module has to be restarted. For the example given in the discussion of fault localization we see that only CAN has to be restarted in order to bring the system into a correct state.

The repair step ensures that components, which are relevant for a certain fault, have to be restarted. This reduces time for bringing a system back to operations. Otherwise, the whole system has to be restarted, which usually takes a much longer time.

Steinbauer et al. also give some initial results in their paper. In particular they consider two situations. In one the Laser Service was killed. In the other the World Model was externally terminated. In both cases the diagnostic process detected the failure, localized the root cause, and brought the robot again to a state where ordinary operation was guaranteed. Note that the authors also reported the use of the system during a RoboCup tournament where a crash related to the image processing module has been solved without human interaction.

From the discussed application on software repair at runtime we are able to draw the following conclusions: (1) The approach allows for effectively autonomous repair of software in case of severe faults like crashes and deadlocks. (2) The use of simple models like the dependency-based model are sufficient in this application domain. (3) The MBD approach can also be used in cases where fault detection via monitoring is separated from fault localization. In this case the monitoring component has to deliver the formal observations. It might be necessary to perform an abstraction step for this purpose where continuous values are mapped to a qualitative value like ok. Note that there are other approaches to software debugging where the program itself is used directly as a model. See for example [53–55]. These approaches can hardly be used directly for runtime diagnosis and repair because of computation requirements. The availability of the proper models is a crucial issue for the acceptance of MBR approaches. For many application the model are handcrafted which is not feasible for larger system or if one do not has access to all internal details of a module. In [56] the authors presented a machine learning approach for model acquisition in the context of robot control software.

4.2 Adaptive Kinematics Control

The second application we discuss is a model-based framework that is able to deal with hardware faults in the driving unit of an autonomous mobile robot. The work presented in [57] focusses on the issue of retaining as much as possible of the robot's functionality when a hardware fault occurs. The proposed framework is self-adaptive in the sense that it follows grateful degradation once a fault is detected. If the system is able to compensate a detected fault it uses model-based reconfiguration. If the system is not able to compensate the fault it reduces the functionality to a lower level that is known and stable. Hence, the knowledge of the degraded system can be used in planning and control afterwards.

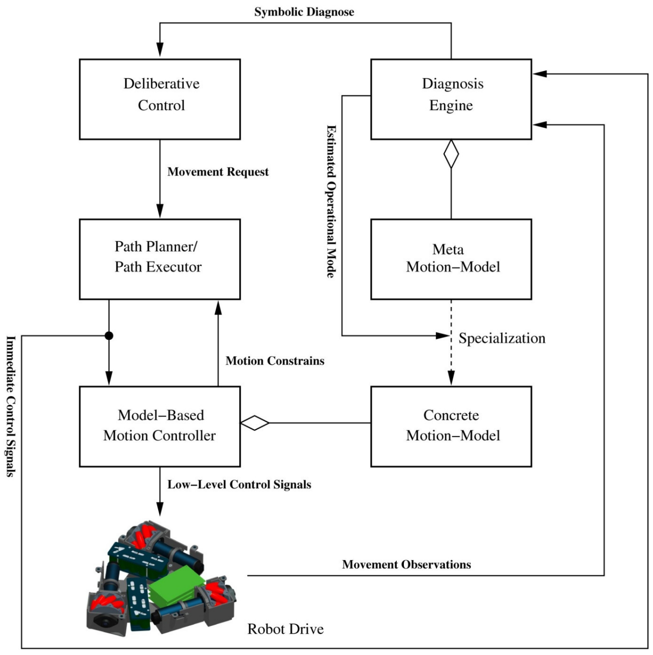


Fig. 6. A adaptive robot control framework

Figure 6 depicts the proposed adaptive robot control framework. The left side of the figure is a classical navigation stack for an autonomous robot [58]. A deliberative control module such as an abstract planner chooses a location where to go next. Such a movement request is communicated down to a path planner, which calculates a collision-free path to the requested goal. Once a feasible path is generated the path executor guides the robot along that path by sending immediate motion commands to the motion controller. The controller converts the motion commands in low-level control commands for the motors, e.g., speed and direction. Such control architectures (abstract control flow) and controllers (dedicated control laws) are usually designed and optimized off-line and do not change during operation. In particular the controller sticks on the kinematics of the robot, the mapping of the control parameter to the robot's motion.

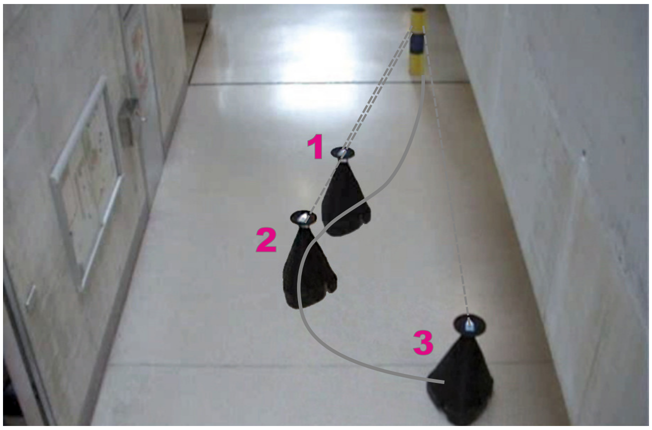
In the case of a hardware fault, e.g., a motor is stuck, the kinematic changes drastically and immediately. If the control architecture is not aware of such changes, it continues its operation and may endanger the mission, the robot, and its environment. This happens because of the changed and now unknown behavior of the system.

In order to allow the robot to react actively to such incidences the control architecture is extended with diagnosis and reconfiguration modules shown on the right side of Figure 6. The basic idea is that the architecture determines the operational mode of the robot drive, i.e., one or more nominal modes and a number of faulty modes, and adapts the model-based controller on-the-fly to reflect the new kinematics that actually rules the robot. As long as the model-based controller is able to adapt to the new kinematics, it simple transparently remaps the commands accordingly. In this case there is no need that the upper modules recognize that there is a fault. If the controller is not able to fully compensate a fault, it degrade the functionality to a lower but known and stable level. Moreover, it informs the upper modules about the degradation in order to allow the path-planner and the deliberative control to adapt their plans to the new situation.

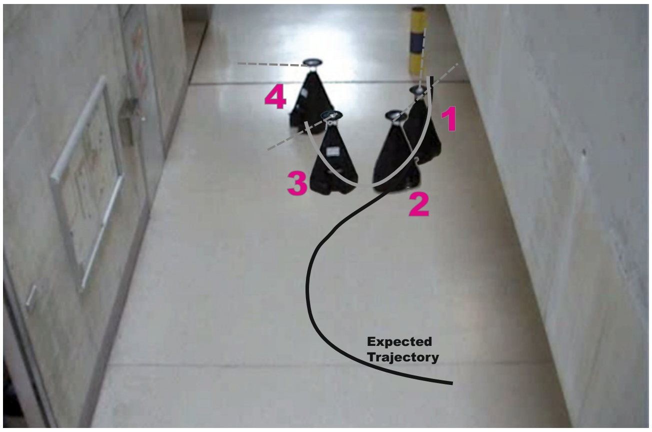
Figure 7 shows a practical example how the adaptive kinematics control works. The example shows an omni-directional autonomous robot executing the so-called *Eye-catcher Task*. For this task the robot has to follow some given path while always looking towards a fixed point (the colored pole on the figures). This task requires omni-directional motion capabilities. The robot has to control all three degrees-of-freedom (DOF) on the plane. Sub-figure (a) shows the correct behavior when the robot works as expected. The robot drives along a s-shape trajectory. The robot is always directed towards the pole (see time steps 1,2 and 3). Sub-figure (b) depicts the faulty behavior when one of the three motors of the robot fail. The fault occurred between time step 1 (still correct) and time step 2 (already a misalignment in the orientation). The control architecture continues its control without awareness of the fault. This leads to an unexpected behavior (time step 3 and 4). Sub-figure (c) shows the robot behavior using the self-adaptive control architecture. The fault was detected but the model-based controller was not able to retain the full functionality. The robot possesses three motors which is the minimum number to provide omnidirectional motion. If one fails the robot has to degrade its functionality. That is exactly what the model-based controller does. It gives up one DOF (direction) and retains the capacities that allows it to follow the trajectory. It is clearly shown that the robot follows the trajectory all the time (time steps 1-4). Only the direction towards the pole is not maintained anymore. Moreover, the path-planner and path-execution engine is informed about the limited maneuverability and adapts the immediate control signals accordingly.

We now discuss the underlying fault detection and localization process as well as the reconfiguration part.

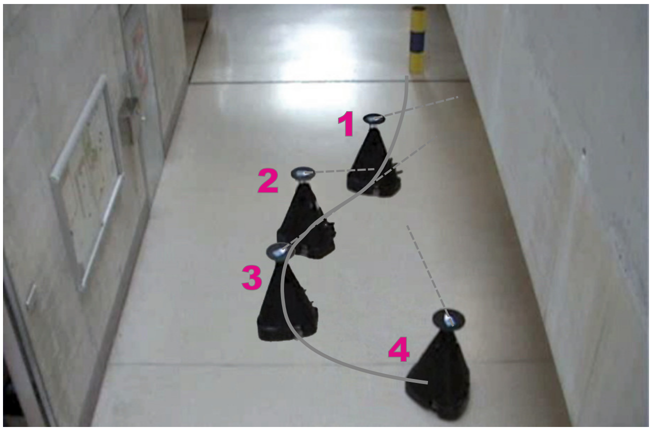
Fault Detection and Localization: Due to the fact that a robot drive is a dynamic system, the actuators and sensors are affected by noise, and the control system has to distinguish several operation modes (different nominal and faulty modes) the fault detection and localization follows the following approach. The behavior of the system is modeled using a hybrid automaton [59]. Figure 8 shows a hybrid automaton for the example part of robot drive depicted in Figure 2. It comprises continuous states, discrete mode states, continuous behaviors for each mode, mode transitions, and probabilities for each transition. The example comprises the discrete modes m_1 (nominal), m_2 (faulty - no torque), m_3 (faulty - stuck), and m_0 (unknown mode). The example transition probability $P_{1,2}$



(a) Correct behavior



(b) Faulty behavior



(c) With self healing

Fig. 7. Adaptive kinematics control in action

is the probability of a change from nominal mode m_1 to faulty mode m_2. The example shows a fully connected automaton. Note that some of the connections can be missing if the transition probability is zero. For instance the probability of a transition from faulty mode m_3 to unknown mode m_0 is quite unlikely. For each discrete mode a continuous behavioral model that relates the continuous input and output values of the system is attached. The continuous input is the voltage applied to the motor u_m. The continuous outputs are the measured rotation velocity ω_e and the current drawn i_c. The continuous behaviors are expressed by differential equations. The equations for mode m_1, m_2 and m_3 are shown in Figure 8. The matrix A comprises constants related to the motor such as its mechanical and electrical properties. The term W_i represents a normal distributed random variable representing the noise in the continuous behavior of mode m_i. Please note that the equation for mode m_3 is no differential equation but simply constraints the rotational velocity to zero. As the unknown mode m_0 comprises no information on the actual behavior no equation is given at all. This does not restrict the variables in any way.

$$\frac{t}{dt}\begin{pmatrix} i_c \\ \omega_e \end{pmatrix} = A\begin{pmatrix} i_c \\ \omega_e \end{pmatrix} + bu_m + W_1; m_1 \tag{7}$$

$$\frac{t}{dt}\begin{pmatrix} i_c \\ \omega_e \end{pmatrix} = A\begin{pmatrix} i_c \\ \omega_e \end{pmatrix} + W_2; m_2 \tag{8}$$

$$\omega_e = 0 + W_3; m_3 \tag{9}$$

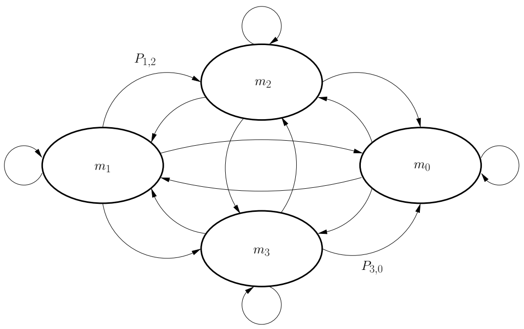


Fig. 8. Hybrid automaton for the example part of a robot drive. The upper part shows the differential equations for the behaviors in the nominal, no torque and stuck mode. The lower part shows the possible mode transitions.

Fault detection and localization is done by estimating the continuous state using the different behavioral models and a multi-hypotheses tracking approach to detect the current discrete mode. The goal is to find the most probable mode the system is in based on the past values of the continuous input and outputs. State estimation for the continuous states is necessary because the system and the measurements are noisy. The multi-hypotheses tracking returns that sequence of discrete modes (changes) that best fit with the continuous state estimation.

If the most probable mode is a faulty mode a fault is detected. The most probable mode gives also information about the fault localization as each fault mode is connected to some fault scenario. It is worth noting that one has to provide a discrete mode, transition probabilities, and a continuous behavioral model for each fault to be detect using the hybrid automaton approach. Faults that are not modeled are caught by an unspecific behavioral model for mode m_0.

Note that the approach follows our given basic definitions of diagnosis using fault modes. The only real difference is that a automaton is used instead of a logical description of the behavior. A mode assignment or in this case a particular state, is a diagnosis if the output is equivalent with the expectations.

Model-based Reconfiguration: Once the most probable operation mode has been identified the adaptive controller architecture starts repair using reconfiguration.The identified state gives information about the faulty components and their behavior, e.g., Motor 3 is stuck. The reconfiguration is achieved by treating the faulty components as additional kinematic constraints. A blocked motor cannot be controlled any more and does not provide any motion. Using this information one is able to determine the actual kinematics of the robot on-line. Moreover, using the actual discrete mode one can derive degradations in capabilities. Like in the *eye-catcher* example above. Because of the geometry of the robot drive and the modes of the motors one can determine whether the robot drive provides full functionality or not. If the functionality degrades (e.g. an omnidirectional robot degrades to a differential robot) this information is communicated to the higher level such as a path planner allowing to reconfigure the path planning to the new kinematics. Even in the case that the fault is very serious and the functionality is completely lost this information is valuable for the higher levels. They may set the robot in a inactive safe state and allow for informing an operator.

5 Conclusions

In the paper we introduced the foundations of model-based reasoning in the context of self-adaptive systems. In particular we present the basic formalization of model-based diagnosis which is founded on earlier work of several colleagues. Moreover, we presented our approach to integrate it into a sense-plan-act architecture of autonomous mobile robots. For this purpose we discussed our extension to a planning and plan execution framework. Moreover, we also discussed two example applications where we used model-based reasoning to increase the robustness of a robot system. In the first application a very abstract dependence model is used to identify failing software components, which are restarted in order to bring the system again into an operational state. The second application makes use of a kinematics knowledge of robot drives. The underlying model captures the behavior using differential equations and explicit fault models. As a result not only faults can be localized but information on degraded behavior can be used to change the control algorithms. There are more applications of model-based reasoning in this domain. We also briefly discussed some of them.

Model-based reasoning provides the necessary foundations for self-adaptive systems. Not only that some of the underlying challenges like diagnosis can be formalized.

The approach also ensures that all results are correct and optimal with respect to the given model. If the model is correct, then the outcome of diagnosis based on the model has to be correct too. Moreover, the approach can not only be used for hardware diagnosis but also for software diagnosis at runtime. The computational overhead of the method is higher but when using simplifying assumptions like that there are only single or double faults, model-based reasoning is feasible. Even larger systems up to several thousand of components can be diagnosed in a reasonable amount of time.

In the domain of autonomous mobile robots using model-based reasoning is a good choice because there the structure of the hardware as well as the behavior of the components are known. Moreover, even fault modes are available for some of the components, which usually make diagnosis more precise. It is worth noting that in cases where no knowledge of the faulty behavior is known, the model-based reasoning approach can still be used. This also distinguishes this approach from others.

Acknowledgments. The work presented in the paper has been funded by the Austrian Science Fund (FWF) under contract number P22690.

References

1. Brooks, R.: Cambrian Intelligence. MIT Press, Cambridge (1999)
2. Pill, I., Quaritsch, T., Wotawa, F.: From Conflicts to Diagnoses: An Empirical Evaluation of Minimal Hitting Set Algorithms. In: Proceedings of the 2nd International Workshop on Principles of Diagnosis (DX 2011), Murnau, Germany (October 2011)
3. Struss, P., Price, C.: Model-based systems in the automotive industry. AI Mag. 24(4), 17–34 (2004)
4. Gat, E.: On three-layer architectures. In: Artificial Intelligence and Mobile Robots. MIT Press (1998)
5. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: 2007 Future of Software Engineering, FOSE 2007, pp. 259–268. IEEE Computer Society, Washington, DC (2007)
6. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secur. Comput. 1(1), 11–33 (2004)
7. Lussier, B., Lampe, A., Chatila, R., Guiochet, J., Ingrand, F., Killijian, M.O., Powell, D.: Fault Tolerance in Autonomous Systems: How and How Much? In: 4th IARP - IEEE/RAS - EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments, Nagoya, Japan (2005)
8. Steinbauer, G.: A Survey about Faults of Robots used in RoboCup. In: Chen, X., Stone, P., Sucar, L.E., der Zant, T.V. (eds.) RoboCup-2012: Robot Soccer World Cup XVI. LNCS (LNAI). Springer, Berlin (2013)
9. Cascio, F., Console, L., Guagliumi, M., Osella, M., Panati, A., Sottano, S., Dupré, D.T.: Generating on-board diagnostics of dynamic automotive systems based on qualitative models. AI Communications 12(1/2) (1999)
10. Malik, A., Struss, P., Sachenbacher, M.: Case studies in model-based diagnosis and fault analysis of car-subsystems. In: Proceedings of the European Conference on Artificial Intelligence (ECAI) (1996)
11. Milde, H., Guckenbiehl, T., Malik, A., Neumann, B., Struss, P.: Integrating Model-based Diagnosis Techniques into Current Work Processes – Three Case Studies from the INDIA Project. AI Communications 13 (2000); Special Issue on Industrial Applications of Model-Based Reasoning

12. Sachenbacher, M., Struss, P., Carlén, C.M.: A Prototype for Model-based On-board Diagnosis of Automotive Systems. AI Communications 13 (2000); Special Issue on Industrial Applications of Model-Based Reasoning
13. Picardi, C., Bray, R., Cascio, F., Console, L., Dague, P., Dressler, O., Millet, D., Rehfus, B., Struss, P., Vallée, C.: Idd: Integrating diagnosis in the design of automotive systems. In: Proceedings of the European Conference on Artificial Intelligence (ECAI), Lyon, France, pp. 628–632. IOS Press (2002)
14. Weld, D., de Kleer, J. (eds.): Readings in Qualitative Reasoning about Physical Systems. Morgan Kaufmann (1989)
15. Pell, B., Bernard, D., Chien, S., Gat, E., Muscettola, N., Nayak, P., Wagner, M., Williams, B.: A remote-agent prototype for spacecraft autonomy. In: Proc. of the SPIE Conference on Optical Science, Engineering, and Instrumentation, New Millennium, Bellingham, Waschington, U.S.A. Space Sciencecraft Control and Tracking, Society of Professional Image Engineers (1996)
16. Williams, B.C., Nayak, P.P.: Immobile robots – ai in the new millennium. AI Magazine, 16–35 (1996)
17. Williams, B.C., Nayak, P.P.: A Model-based Approach to Reactive Self-Configuring Systems. In: Proceedings of the Seventh International Workshop on Principles of Diagnosis, pp. 267–274 (1996)
18. Williams, B.C., Nayak, P.P.: A reactive planner for a model-based executive. In: Proceedings 15th International Joint Conf. on Artificial Intelligence, pp. 1178–1185 (1997)
19. Crow, J., Rushby, J.: Model-based reconfiguration: Toward an integration with diagnosis. In: Proceedings AAAI, pp. 836–841. Morgan Kaufmann, Los Angeles (1991)
20. Stumptner, M., Wotawa, F.: Model-based reconfiguration. In: Proceedings Artificial Intelligence in Design, Lisbon, Portugal (1998)
21. Ardissono, L., Furnari, R., Goy, A., Petrone, G., Segnan, M.: A soa-based model supporting adaptive web-based applications. In: Proc. of 3rd Conference on Internet and Web Applications and Services (ICIW 2008), pp. 708–713. IEEE, Athens (2008)
22. Cordier, M.O., Pencolé, Y., Travé-Massuyés, L., Vidal, T.: Characterizing and checking self-healability. In: Proc. of 18th European Conference on Artificial Intelligence (ECAI 2008), Patras, Grece (July 2008)
23. Karsai, G., Abdelwahed, S., Biswas, G.: Integrated diagnosis and control for hybrid dynamic systems. In: AIAA Guidance, AIAA Guidance, Navigation and Control Conference, Austin, Texas (August 2003)
24. Narasimhan, S., Biswas, G., Karsai, G., Szemetzy, T., Bowman, T., Kay, M., Keller, K.: Hybrid modeling and diagnosis in the real world: A case study. Working Papers Thirteenth Int Workshop Principles of Diagnosis (June 2002)
25. Giese, H., Henkler, S., Hirsch, M.: A multi-paradigm approach supporting the modular execution of reconfigurable hybrid systems. Simulation 87(9), 775–808 (2011)
26. Davis, R., Shrobe, H., Hamscher, W., Wieckert, K., Shirley, M., Polit, S.: Diagnosis based on structure and function. In: Proceedings AAAI, Pittsburgh, pp. 137–142 (August 1982)
27. Davis, R.: Diagnostic reasoning based on structure and behavior. Artificial Intelligence 24, 347–410 (1984)
28. Davis, R., Hamscher, W.: Model-based reasoning: Troubleshooting. In: Shrobe, H.E. (ed.) Exploring Artificial Intelligence, pp. 297–346. Morgan Kaufmann (1988)
29. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence 32(1), 57–95 (1987)
30. de Kleer, J., Williams, B.C.: Diagnosing multiple faults. Artificial Intelligence 32(1), 97–130 (1987)

31. Wotawa, F.: Adaptive Autonomous Systems – From the System's Architecture to Testing. In: Hähnle, R., Knoop, J., Margaria, T., Schreiner, D., Steffen, B. (eds.) ISoLA 2011 Workshops 2011. CCIS, vol. 336, pp. 76–90. Springer, Heidelberg (2012)
32. Steinbauer, G., Weber, J., Wotawa, F.: From the real-world to its qualitative representation–practical lessons learned. In: International Workshop on Qualitative Reasoning, pp. 186–191 (2005)
33. de Kleer, J., Mackworth, A.K., Reiter, R.: Characterizing diagnosis and systems. Artificial Intelligence 56 (1992)
34. Console, L., Dupré, D.T., Torasso, P.: On the relationship between abduction and deduction. Journal of Logic and Computation 1(5), 661–690 (1991)
35. Friedrich, G., Gottlob, G., Nejdl, W.: Physical impossibility instead of fault models. In: Proceedings of the National Conference on Artificial Intelligence (AAAI), pp. 331–336, Boston (August 1990); Also appears in Readings in Model-Based Diagnosis. Morgan Kaufmann (1992)
36. Greiner, R., Smith, B.A., Wilkerson, R.W.: A correction to the algorithm in Reiter's theory of diagnosis. Artificial Intelligence 41(1), 79–88 (1989)
37. de Kleer, J.: An assumption-based TMS. Artificial Intelligence 28, 127–162 (1986)
38. Fröhlich, P., Nejdl, W.: A Static Model-Based Engine for Model-Based Reasoning. In: Proceedings 15th International Joint Conf. on Artificial Intelligence, Nagoya, Japan (August 1997)
39. Fattah, Y.E., Dechter, R.: Diagnosing tree-decomposable circuits. In: Proceedings 14th International Joint Conf. on Artificial Intelligence, pp. 1742–1748 (1995)
40. Stumptner, M., Wotawa, F.: Diagnosing tree-structured systems. Artificial Intelligence 127(1), 1–29 (2001)
41. Sachenbacher, M., Williams, B.C.: Diagnosis as semiring-based constraint optimization. In: Proceedings of the 16th European Conference on Artificial Intelligence (ECAI), Valencia, Spain, pp. 873–877 (2004)
42. Weld, D.S.: Recent advances in ai planning. AI Magazine 20(2), 93–123 (1999)
43. Russel, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Prentice-Hall (2010)
44. Sun, Y., Weld, D.S.: Beyond simple observation: Planning to diagnose. In: Third International Workshop on Principles of Diagnosis, Rosario, (WA) (October 1992)
45. Friedrich, G., Gottlob, G., Nejdl, W.: Towards a theory of the repair process. In: Proceedings of the Portuguese Conference on Artificial Intelligence. LNCS (LNAI), Springer, Albufeira (1991); Also appeared at the Model-Based Reasoning Workshop (AAAI 1991), Anaheim (July 1991)
46. Friedrich, G., Gottlob, G., Nejdl, W.: Formalizing the repair process. In: Proceedings of the European Conference on Artificial Intelligence (ECAI), pp. 709–713. John Wiley & Sons, Vienna (1992); Also appeared in the Proceedings of the Second International Workshop on Principles of Diagnosis, Milano (1991)
47. Friedrich, G., Nejdl, W.: Choosing observations and actions in model-based diagnosis-repair systems. In: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning, Cambridge, MA (October1992)
48. Wotawa, F., Nica, M., Aichernig, B.K.: Generating distinguishing tests using the minion constraint solver. In: CSTVA 2010: Proceedings of the 2nd Workshop on Constraints for Testing, Verification and Analysis. IEEE (2010)
49. Steinbauer, G., Mörth, M., Wotawa, F.: Real-time diagnosis and repair of faults of robot control software. In: Proceedings RoboCup International Symposium (2005)
50. Friedrich, G., Stumptner, M., Wotawa, F.: Model-based diagnosis of hardware designs. Artificial Intelligence 111(2), 3–39 (1999)

51. Wotawa, F.: On the Relationship between Model-Based Debugging and Program Slicing. Artificial Intelligence 135(1-2), 124–143 (2002)
52. Weiser, M.: Programmers use slices when debugging. Communications of the ACM 25(7), 446–452 (1982)
53. Mayer, W.: Static and hybrid analysis in model-based debugging. PhD Thesis, School of Computer and Information Science, University of South Australia (2007)
54. Nica, M., Nica, S., Wotawa, F.: On the use of mutations and testing for debugging. In: Software: practice & experience (in Press, 2012), `http://dx.doi.org/10.1002/spe.1142`
55. Wotawa, F., Nica, M., Moraru, I.D.: Automated debugging based on a constraint model of the program and a test case. The Journal of Logic and Algebraic Programming (in Press, 2012)
56. Kleiner, A., Steinbauer, G., Wotawa, F.: Towards Automated Online Diagnosis of Robot Navigation Software. In: Carpin, S., Noda, I., Pagello, E., Reggiani, M., von Stryk, O. (eds.) SIMPAR 2008. LNCS (LNAI), vol. 5325, pp. 159–170. Springer, Heidelberg (2008)
57. Brandstötter, M., Hofbaur, M.W., Steinbauer, G., Wotawa, F.: Model-based fault diagnosis and reconfiguration of robot drives. In: 2007 IEEE/RSJ International Conference on Intelligent Robots and System, pp. 1203–1209 (2007)
58. Murphy, R.R.: Introduction to AI Robotics, 2nd edn. MIT Press, Cambridge (2002)
59. Hofbaur, M.W., Williams, B.C.: Mode Estimation of Probabilistic Hybrid Systems. In: Tomlin, C.J., Greenstreet, M.R. (eds.) HSCC 2002. LNCS, vol. 2289, pp. 253–266. Springer, Heidelberg (2002)

Achieving Self-adaptation through Dynamic Group Management

Luciano Baresi, Sam Guinea, and Panteha Saeedi

Politecnico di Milano, Dipartimento di Elettronica e Informazione
via Golgi, 42 – 20133 Milano, Italy
{baresi,guinea,saeedi}@elet.polimi.it

Abstract. Distributed pervasive systems have been employed in a wide spectrum of applications, from environmental monitoring to smart transportation, to emergency response. In all these applications high volumes of typically volatile software components need to coordinate and collaborate to achieve a common goal, given a defined set of constraints.

In our A-3 initiative we advocate that the coordination of high volumes of volatile components can be simplified using appropriate group abstractions. Instead of attempting to coordinate large amounts of components, the problem can be reduced to coordinating "groups" of components which have a less dynamic behavior. This abstraction effectively simplifies the design of self-adaptive behavior by making it easier to achieve component coordination. By design it also prevents the system from being flooded with coordination messages.

In this chapter we present an extension of our A-3 middleware called A3-TAG. It is a unified programming model that facilitates the system design. Moreover, the middleware internally adopts the same group abstractions to ensure that message exchanges, and in particular group broadcasts, are achieved efficiently and robustly. The chapter also presents an investigation of our approach in the context of a self-adaptive industrialized greenhouse.

1 Introduction

Distributed and ubiquitous systems have been employed in a wide spectrum of applications, such as environmental monitoring [1], wildlife tracking [2], intelligent agriculture [3], home automation [4], building monitoring and control [5], smart transportation [6], tracking and surveillance [8], and emergency response [12]. In all these applications several software components need to collaborate to achieve a common goal. Traditionally, many of these distributed systems are coordinated using some kind of data-gathering capability, the design of which, in most cases, is closely-coupled with the actual application logic [11]. This makes the systems very inflexible and scarcely scalable: important changes, when needed, often end up requiring some form of re-coding.

We advocate that these capabilities cannot be dealt with at the application level; a suitable middleware infrastructure must provide convenient abstractions to ease the problem. Moreover, the next generation of distributed, ubiquitous systems, referred to by some as *felicitous* systems [13], will be even more demanding, and will require that interactions among components be carried out in a way that is optimal and naturally suited to the circumstances and needs of the users. The middleware infrastructure should not only be able to present a unified programming model, and suitable abstractions to ease the realization of the system, but it should also be able to perform intelligent self-adaptation.

Self-adaptation is the capability of a system to autonomously make decisions on how it should react to changes in the context, or in its functional or non-functional requirements. However, to implement a self-adaptive distributed system, the components need to be able to share their knowledge and coordinate their reactions in an effective and robust way.

A3-TAG (A3 with Tuple-space Assisted Grouping) is an extended version of our middleware infrastructure A-3 [14,15]. It has been augmented with a tuple space to ease the coordination among components [7], and through this coordination facilitate the development of trustable self-adaptive systems.

A3-TAG simplifies the coordination of distributed, ubiquitous systems through the *group* abstraction. Instead of attempting to coordinate very large amounts of components, it reduces the problem to the coordination of "groups" of components, which usually are fewer and have a less dynamic behavior. Components can be grouped together for very different reasons: for example, because they are conceptually similar, have common goals in the context of the application, or are physically close one to the another. Each group is managed by a *supervisor*, which is in charge of the interactions with the other groups (i.e., the other supervisors).

This chapter provides a thorough presentation of A3-TAG. The tuple-space assisted grouping enables a stricter separation of concerns between the system's application logic and its group management, providing higher assurances that the middleware will be able to autonomously perform group configuration, discovery, monitoring, and recovery. The introduction of the tuple space also allows components in the system to delegate state management directly to the middleware. State is maintained in a distributed fashion, allowing the system to re-construct itself easily and rapidly when a key component fails. For example, a failing supervisor can easily be replaced, and the new supervisor will be able to immediately start where the old one left off.

The key features of A3-TAG are exemplified on a running example based on a self-adaptive greenhouse. Various sensors are spread out through the greenhouse, allowing the system to monitor the greenhouse's temperature, humidity, and levels of CO_2. The greenhouse also has a certain number of fans and irrigation systems to use when the sensed values drift outside the pre-defined range. The example shows how group management can minimize the amount of traffic being generated while the greenhouse is monitored, and shows how the groups can evolve to cover the evolution of the greenhouse (the plants and flowers it contains) and the situations in which key components in the system fail.

The rest of the chapter is organized as follows. Section 2 surveys some related proposals. Section 3 describes the basic concepts behind A3-TAG and clarifies the separation of concerns that exists between application logic and group management. Section 4 explains how the tuple space is used as groups evolve over time. Section 5 exemplifies the behavior of our middleware in the self-adaptive greenhouse, Section 6 provides an evaluation of our A3-TAG middleware, and Section 7 concludes the chapter.

2 Related Work

There is a substantial amount of literature on reconfigurable middleware systems that focus on problems such as the self-adaptation of the system's components [37], others on data sharing [21], still others on spontaneous and opportunistic coordination [34]. The approaches that have goals that are nearer to A3-TAG are those that concentrate on automatic component configuration. Projects like *Gaia* [35] provide a middleware for automatic configuration in Smart Environments. These kinds of systems represent highly integrated environments and support various stationary and mobile devices. However, they are not designed to function in ad hoc environments, as they rely on an existing infrastructure. Another example, the RUNES component-based middleware [12], only targets wireless ad hoc networks on embedded systems. Its goal is to enable scenarios such as emergency response in which devices need to exhibit coordinated behavior. The approach disseminates the configuration tasks equally among all devices, and does not exploit resource-rich devices in heterogeneous environments (i.e. clustering and re-clustering). In order to overcome the limitations of these approaches, Schuhmann et al. [22] investigated a *Hybrid approach* based on the formation of clusters with balanced configuration loads for resource-rich devices. These devices represent the "active devices", while the resource-weak devices remain passive to avoid bottlenecks. The hybrid approach automatically adjusts its degree of decentralization with respect to the available resources in the network. This strongly reduces the configuration time and helps increase users' acceptance of ubiquitous systems; it represents a large step towards automatic application configuration. Similar ideas have been applied in the context of the A3-TAG middleware. A first difference, however, is that A3-TAG is an innovative solution based on *group abstraction* that can ideally be used in any kind of dynamic distributed systems. Second, the reason why A-3 groups its components is not limited to load balancing; instead, policies can be application-specific and are dynamically configurable.

Other interesting works that have similar goals to ours are those that have flourished in the context of Organic Computing. Research in organic computing studies nature to learn how it deals with self-$*$ properties. Roth et al. [38] have presented the $OC\mu$ middleware. It is a service-oriented middleware for smart environments. *Organic Managers* are assigned to each node in the system to provide sophisticated monitoring and self-$*$ property management. They focus on providing self-configuration (i.e., initial service distribution under constraints), self-optimization (i.e., runtime balancing based on an artificial hormone system), self-protection, and self-healing (i.e., recovery planning). von Renteln et al.[39] [40] have a similar approach that focuses on self-organization and self-healing. They have developed an AHS (Artificial Hormone System) for mapping computational tasks onto a system's processing elements. Their approach can be used under real-time constraints, and provides features such as partial suppression of tasks and distributed task termination. Although these bio-inspired works have similar goals to ours, they are profoundly different in their approach. They have proven through simulation to be scalable, robust, and capable of handling malfunctioning components, yet they lack proper design approaches that support their application. We, on the other hand, focus on developing a novel architectural style, and the appropriate abstractions, to ease the design of such systems as much as possible.

Another thriving area of research is that of multi-agent systems for self-adaptive systems. In Weyns et al.'s works [41] a situated multi-agent system is structured as a set of interacting autonomous entities that are situated in an environment. They employ the environment to share information and coordinate their behavior in a decentralized fashion. In their work an agent is capable of *perception* (i.e., a filtered sensing of the environment), *decision making* (i.e., action selection through an influence-reaction model), and *communication* (i.e., interaction with other agents). In subsequent work [42], the authors used these agents to develop bio-inspired patterns for delegate MAS (Multi-Agent Systems). They developed three bio-inspired and light-weight patterns to tackle the problems of global-to-local and local-to-global information dissemination, and of stability, i.e., how to limit the continuous system revisions that may be due to arising possibilities and problems. These works assume a completely decentralized solution, with all the design complications that this implies. Instead, the A-3 initiative strives to develop abstractions for reasoning on a decentralized system, without loosing the advantages of having components that play more "central" roles in the self-adaptation. A-3 also provides a clearer notion of composition, which we believe allows the designer to more easily come up with global solutions through simpler local reasoning.

Another important work in the field of multi-agent systems was provided by Shehory et al. [43]. In their work they proposed to coordinate multiple agents through *distributed coalition formation.* Coalition formation allows a system to optimally tackle multiple incoming task requests through the prioritization of these tasks, since in real-world scenarios it may not be even possible to accomplish them all. In this work, each agent has a set of quantifiable capabilities, and each task is described by the capabilities that are required for it to be performed. Moreover, each completed task produces a quantifiable benefit. A *coalition* is a group of agents that group together to perform a common task. Agents join groups if the joint benefits of the tasks they are able to complete are at least equal or greater than the benefits they would receive by remaining outside of the groups. This work differs from ours because it formalizes why certain agents/components should group together in a system, while we focus on the assurances they have once they are grouped. In a sense the works can be considered complementary, and we are considering further investigation this in our future work.

Coordination models that are based on message passing (e.g., Publish and Subscribe) are also of general interest to our work. In these coordination models data is typically available only at the moment it is published. Generally speaking, most approaches based on Publish and Subscribe [25] require the simultaneous presence of all the coordinated parties. Solutions such as lease mechanisms [23] have been proposed; they can tolerate message loss, however, they also consume a considerable amount of network resources. In literature there have been investigations on how to implement data sharing coordination on top of message passing techniques [21,24,18,26] to overcome this problem. A detailed analysis of relevant approaches, however, shows that there are still major recurring limitations regarding system flexibility and scalability. In order to manage the diversity and dynamics of such systems, a middleware needs to be able to self-adapt and disseminate new configuration tasks depending on the needs and availability of its components. The A3-TAG initiative is being developed with these two goals clearly in mind.

Finally, many approaches in the field of self-organizing peer to peer (P2P) networks focus on means to achieve reasonable performance by avoiding message overhead and congestion, such as [36] and [44]. These techniques, however, have been shown to be costly and inefficient with high churn rates. One way to overcome these problems has been to introduce group-based solutions. Similarly to A-3, these choose a number of key peers, that are less prone to suddenly enter or leave the system, and use them to implement the management duties. Scope-based approaches [45] connect peers that have common interests, yet this is not always feasible with mobile peers that have limited resources. Location-based approaches [46] [47] and proximity-based approaches [49] [48] [50] attempt to solve these problems by taking into account the peers' locations and concepts such as multi-hop forwarding. Nevertheless, these protocols are still not highly scalable, since they involve all the peers and risk flooding the network. Our approach is a group-based approach that pushes research in this area forward by exploiting a more rational distribution of the key peers (i.e., the supervisors) to obtain greater optimization.

3 A3-TAG's Main Concepts

A3-TAG simplifies the coordination of distributed systems through the notion of *group*. Instead of attempting to coordinate a very large group of components, which may enter or leave the system freely, we reduce the problem to the coordination of groups of components, which are fewer and less dynamic in their behavior. Therefore, a group reunites multiple components that we assume can be coordinated as a unique identity.

When a component enters a group, it can either be a *supervisor* or a *supervised* component. By definition a group can have only one supervisor, and the first component to enter the group must assume this role, while it can have as many supervised components as needed. This allows coordination to be achieved on a smaller scale. Indeed, supervised components can use intra-group communication to pass on information to their supervisor(s), so that is can digest the data it receives and push back any behavioral directives it may have.

The complexity that lies in coordinating the entire system is managed by composing such groups. This is achieved by allowing components to belong to more than one group at a time, and to allow them to play different roles in different groups.

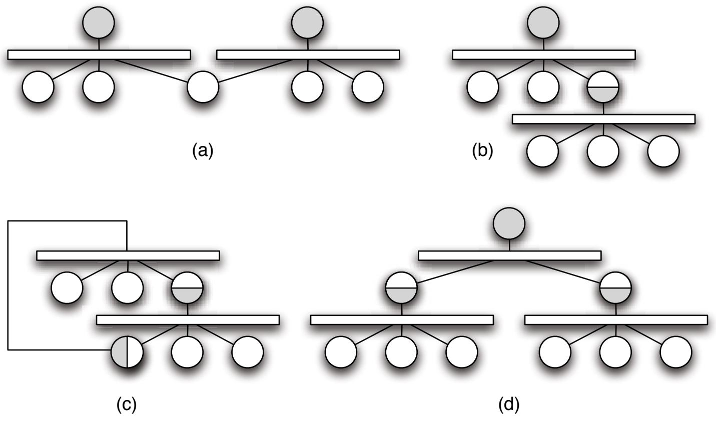


Fig. 1. Group composition

Figure 1 illustrates how two groups can be composed. White circles represent supervised components, while grey circles represent supervisor components. Two-colored circles represent components playing two different roles in two different groups. In Figure 1(a) we have a supervised component that simultaneously belongs to two groups. This means that it sends its information to two different supervisors. It is up to the supervised component to harmonize any contrasting directives that might come from its supervisors. Figure 1(b) introduces hierarchical composition. A top group is supervising three components. However, one of these components is actually hiding the existence of a second "nested" group. This component contributes to the top group with a digest of the knowledge it collects from the bottom group. This allows the top group's supervisor to have a complete view of the system, without having to interact with all the components in the system. Figure 1(c) extends this composition by making it bi-directional. Both supervisors have a complete view of the system. Finally, Figure 1(d) shows a more "classic" hierarchical solution for coordination. We have two bottom-level groups, and each group's supervisor is part of a higher-level group through which they can be coordinated. The higher-level group's supervisor is, at least in this figure, a centralized coordinator.

3.1 Components

A3-TAG alleviates the development task by allowing the developer to focus on the application logic that the various components in the system need to provide. All group management is delegated to the A3-TAG middleware, and is transparent to the application developer.

When developing an A3-TAG application, the first step is to define the groups that need to exist within the system. This is achieved by describing each group with a XML map of application-specific keys and values. The second step is to define the components that will participate in the system. Components are also described through application-specific maps, so that the middleware can automatically map components to groups.

The A-3 style states that each component can belong to multiple groups. For each group it can either play the role of the supervisor, or the role of a supervised node. For each role, in each group, the developer is required to provide a behavioral implementation. When the middleware places a component inside a group, it tells the component which of its behaviors it needs to activate. If the component is the first component to join a group, it will activate the supervisor component behavior for that group; if not, it will activate the corresponding supervised component behavior.

When developing a component's behavior, the developer can access three important features provided by A3-TAG: *communication*, *views*, and *state management*. While communication is important both for supervisors and supervised components, views and state management are currently only provided to supervisors.

The middleware offers advanced *communication* capabilities based on asynchronous messaging. Supervisors can send messages to their supervised components using broadcast, multicast, and unicast messages. In the first case the message is sent to all the supervised components that are inside the group. In the second case the message is sent to a subset of the supervised components. The subset is determined using component descriptor matching. In practice, if a supervised component's descriptor map contains a desired set of values, it will receive the message. In the last case the message is sent to a single supervised component, which is determined through its unique identifier. In our current implementation the identifier is the component's IP address. Supervised components, on the other hand, can only send messages to the supervisor of their group.

All messaging techniques in A3-TAG provide the following assurances. First of all, messaging is reliable and communication from the supervisor to the supervised nodes is provided with *virtual synchrony*. Virtual synchrony ensures that the messages are delivered to all the intended recipients, and in the correct order. Second, messaging can be temporarily delayed for a recipient if that recipient has momentarily disconnected from the group. The delay is only successful if the recipient returns to the group before the message's timeout expires.

The middleware also offers *views*. This allows supervisor components to understand how many, and what, components are in its group at any given time. This can be a vital piece of information when determining its own application logic.

Finally, the middleware offers *state management*. This allows supervisor components to save arbitrary data to the middleware, where they are kept redundantly, so that they can be recovered at a later time. This simplifies supervisor substitution when there is a supervisor failure, since it allows the new supervisor to more easily continue from where the previous supervisor left off, with minimum downtime.

3.2 Assurances Provided by A3-TAG

Despite the many proposed solutions to design and implement self-adaptive systems, the actual assurances provided by these approaches have been often neglected. Self-adaptive systems can provide effective solutions to tackle real problems if one can trust their capability to coordinate the distributed components' reactions to the changes in the context or in the requirements.

A3-TAG tackles assurances at both the architectural and application level. It is based on *group-based communication*, and we provide an implementation that guarantees both *virtual synchrony* and *delayed communication*. We also provide *state management* through a distributed and reliable tuple space. This means that vital application data can be stored and later recovered, even after the component that originally stored the data has left the system. Transparent group management lets developers assume that group membership is alway correct and accurate: there is no need to waste resources at the application level to manage who the members of a group are.

These low-level guarantees, along with groups, can then be exploited to provide higher-level ones. For example, reliability and robustness can be provided through special-purpose groups. Instead of having a single supervisor per group, one may think of defining special-purpose groups of supervisors, containing nodes that are solely created to be available and ready as backups. The tuple space ensures that all these nodes have access to the same information, and thus as soon as the official supervisor disappears, one of its deputies can easily substitute it and keep the group on track.

Groups can also be exploited to force particular behaviors onto some nodes of the system. Grouping nodes with similar characteristics and behaviors becomes very natural, and the boundaries provided by the group become a natural border against unforeseen behavioral changes. Moreover, groups can help manage context changes. Indeed, the rules that govern the creation of groups can change dynamically, and thus they can transparently accommodate changes in the contexts of operation. Finally, nodes can come and go, meaning that they can enter and/or leave contexts freely. This gives designers a lot of flexibility when they develop their self-adaptive systems.

4 Inside the A3-TAG Middleware

A3-TAG[1] fosters a strong separation of concerns between the application's logical layer, in which designers define the groups and the components that make up the system, and the group management that occurs transparently inside the middleware. This section gives the reader some insights into what actually occurs within the middleware, and how it self-adapts to ensure efficient group management.

Internally the middleware maintains a topology of so-called *coordination groups*. This topology is initially identical to the group topology defined at the application layer. Indeed, for every *application group* the middleware starts a corresponding *coordination group*. As the system evolves, however, the initial coordination group evolves to become a complex topology of multiple smaller coordination groups. This is done to optimize the number of active communication links that are needed to ensure correct and efficient messaging inside the corresponding application group, and is completely transparent to the application developer.

Coordination groups, just like their application-level counterparts, exploit the main A-3 abstractions. Each coordination group has a supervisor and a set of supervised components, and coordination groups are composed by allowing components to belong to more than one coordination group at a time. The supervisors are automatically managed so that the coordination group topology never gets disconnected.

Coordination group topology management is achieved by the middleware mainly to prevent message congestion when there is a sudden increase in the number of components in a coordination group. If the number of components in a coordination group exceeds a configurable threshold, the middleware reacts by splitting the group into two smaller ones, and by connecting them hierarchically to ensure that no message being sent at the application level is ever lost. This solution is activated immediately, in order to attempt to solve the congestion problem as soon as possible. However, this is done at the cost of having to manage multiple coordination groups to ensure the consistency of the corresponding application level group. This is why the system will also periodically attempt to contract the topology of coordination groups by reducing the number of groups being used. This is done if the system can remove some of the groups and move their orphan components into another group, without causing congestion threshold values to be exceeded. As the coordination topology adapts, the coordination group that contains the application group's supervisor component is called the *master* coordination group.

Our middleware [27] is built as an extension to REDS [28], a distributed publish and subscribe middleware. REDS implements distributed algorithms that support the dynamic reconfiguration of its infrastructure, making it resilient to

[1] The A3-TAG middleware is currently available for download at `http://code.google.com/p/rtag/`, together with example code, and the configurations we used to evaluate its performance.

changes in the infrastructure's own topology. A3-TAG exploits REDS for group communication and for managing the topology of groups, and adds a shared memory space under the form of a distributed tuple space that keeps track of how the topologies of application and coordination groups change as components enter or leave the system. This is instrumental in how the middleware searches for a group in which to add a new component, in how it decides to create a new group and compose it with the rest of the system, and in finding the correct destinations for when application messages are sent. It is also instrumental in providing the views and the state management features discussed in the previous section.

The tuple space is implemented using the LIGHTS framework [32]. LIGHTS provides the traditional Linda [30] tuple space abstractions, and the modularity and encapsulation provided by its object-oriented design gave us ample room for customization. In LIGHTS, processes communicate through a shared tuple space that acts as a repository of elementary data structures, or tuples. Each tuple is a sequence of typed fields. They are added to a tuple space by performing an `out(t)` operation, and can be removed by executing `in(p)`. They are anonymous, thus their selection takes place through pattern matching on their content. The argument `p` is a pattern whose fields can contain either actuals or formals. Actuals are values, while formals act like "wild cards". If multiple tuples match a template, the tuple space selects one non-deterministically. Tuples can also be read from the tuple space by using the non-destructive `rd(p)` operation. Both `in` and `rd` are blocking, that is, if no matching tuple is available in the tuple space the process performing the operation is suspended until a matching tuple becomes available. A typical extension is to include a pair of asynchronous primitives `inp` and `rdp`, called probes, that allow non-blocking access to the tuple space.

Our tuple space contains four different kinds of tuples. *Application group tuples* are tuples that are used to keep track of the groups that exist at the application level. Each tuple contains a description of the group's supervisor, and descriptors for the components that exist in that group. *Communication tuples* contain the messages that have been sent within the system, and that are waiting to be removed when their timeout value is up. *Coordination group tuples* are conceptually similar to the application group tuples. The middleware keeps one tuple for each coordination group, and uses them to keep track of the internal coordination topologies. Each tuple contains a description of the group, as well as descriptions of its components. *State management tuples* are used to store information that the application layer has decided to replicate, and make available to components that need to replace a failing, or leaving, supervisor.

As previously stated, the middleware will self-adapt its coordination groups to optimize the message exchanges that occur within the system. In our current implementation this occurs as a reaction to one of two possible situations: either a group has become congested, or the topology of coordination groups gets disconnected. The middleware will also perform periodical re-organization to optimize the internal topology.

Congestion occurs when there are too many components in a group. This is detected by the middleware, which is continuously keeping track of the nodes that enter or leave the system in order to keep the application and coordination group tuples in the tuple space up-to-date. When the middleware sees that a coordination group has grown beyond the pre-defined threshold, it attempts to move the excess nodes to another coordination group inside the same topology.

The middleware starts by looking up one group in the hierarchy, if possible, to see whether it can send over some nodes. Access to the upper group is guaranteed by the fact that the congested group's supervisor is also participating in that group, and, therefore, has access to its coordination group tuple. If there is no room in the upper group, the supervisor asks its supervised components if any of them is leading a sub-group, and if they have room for extra components. If there are no sub-groups, or they have no room, the supervisor asks one of its supervised components to create a new sub-group. In general the strategy is to attempt to move nodes not "too far away" from the congested group. Moving them too far away would be in contrast with the "cleaning up" that the middleware will later attempt to contract the topology. All the corresponding coordination tuples in the tuple space are updated.

More details as to the advantages that the topological re-organization brings about are provided in Section 6.

Coordination Group Supervisor Failure means that the corresponding coordination group gets momentarily disconnected from the coordination topology. This is immediately caught by the middleware, which attempts to reconnect the coordination group by finding a substitute supervisor. The middleware contacts all the components in the group and asks each to evaluate a fitness function. The component with the highest fitness function becomes the new supervisor. A3-TAG provides a default fitness function that takes into account the resources that are available at that time to each component (e.g., battery life). However, we also allow system designers to override the fitness function to take into account application-specific or context-specific runtime information.

When a new supervisor has been chosen, it looks into the tuple space to see where its coordination group was previously attached in the topology. In practice, it looks at the description of the previous supervisor, which contains the other groups in which the supervisor was participating. All the corresponding coordination tuples in the tuple space are updated accordingly.

Periodical Re-Organization looks at the coordination tuples inside the tuple space, and finds two groups that can be merged. In particular, the system analyzes the topology and merges groups that are far from the master group with ones that are closer.

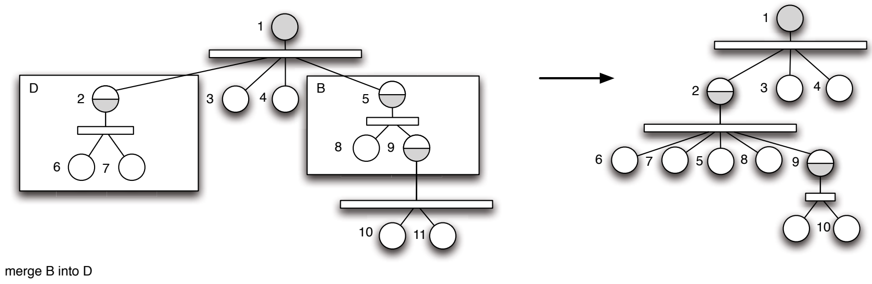


Fig. 2. Group merge

4.1 Adapting the Coordination Group Topology

All the changes in the coordination topology are performed through low-level group operations such as group creation, discovery, merging, deletion, splitting, and division.

Creation and Discovery. When a component enters the system the middleware looks at its application-specific description and attempts to find one or more coordination groups in which to place it. If there are no groups that can accept the component, the component must create its own application group and become its supervisor. This will result in the creation of the coordination master group as well. If the component is not capable of performing as a supervisor, it cannot join the system until an appropriate group appears. If two components execute this procedure simultaneously, they might accidentally create two application groups matching the same query. However, this is not a problem since the middleware will proceed to merge them if.

Merges and deletions. When the middleware needs to merge two groups, it deletes one of the groups and moves all its components to the remaining one. Figure 2 shows an example in which group B is merged into group D: group B is deleted and its components are moved to group D. Group B contains three components: component number 5 is its supervisor, while components 8 and 9 are supervised components. These three become supervised components in group D, and start being managed by supervisor component 2. Component 9, however, is also a supervisor in a third group; moving component 9 to group D causes the entire group to be moved "beneath" group D. The coordination tuple describing group B is removed, and the coordination tuple describing group D is updated to reflect its new configuration. The coordination tuple describing the group managed by component 9 is also modified to reflect its new place in the hierarchy.

Figure 3 illustrates how the middleware can delete groups. Every time we delete a coordination group, a number of components become orphans, and the middleware needs to find suitable groups in which to place them. When the middleware deletes a group inside a hierarchy, the deletion does not propagate to the sub-groups. In the example, the middleware only deletes group B. Components 4, 5, and 6 are all moved into the group managed by component 1, and the group managed by component 6 is left intact. In the tuple space the coordination tuple pertaining to group *B* is deleted, and the coordination tuples describing the groups managed by components 1 and 6 are updated accordingly.

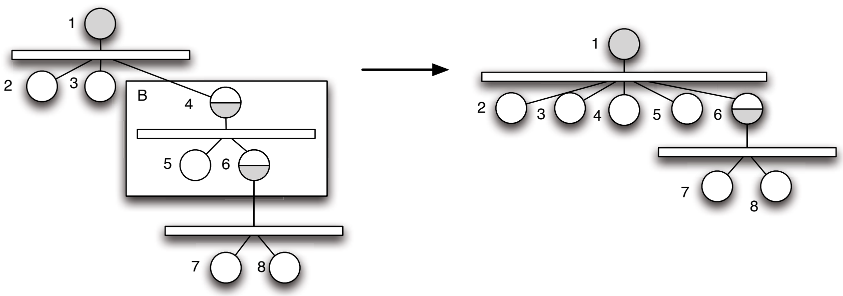


Fig. 3. Group deletion

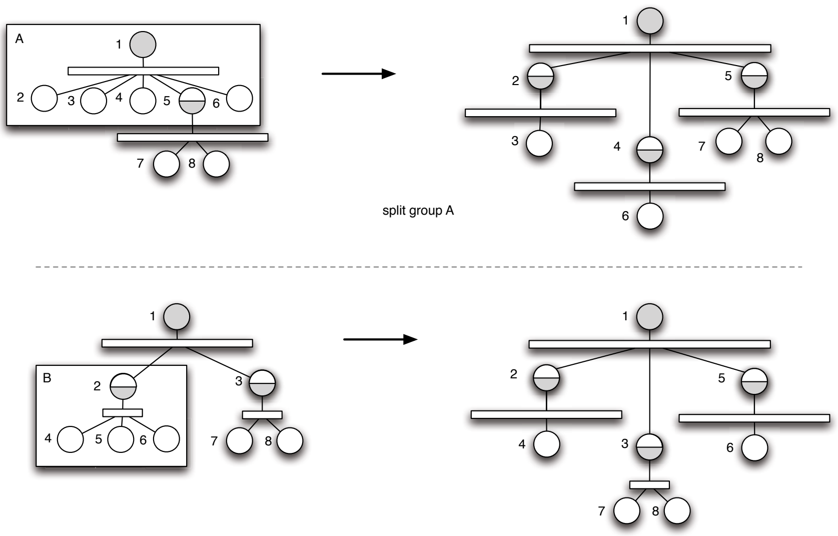


Fig. 4. Group split and division

Splits and Divisions. Splits and divisions cause new groups to stem from existing ones. When the middleware splits a group it takes all the group's supervised components and uses them to create *n* sub-groups that are added hierarchically "beneath" the original group being split. In the top example of Figure 4, the middleware splits group *A* into three sub-groups. One sub-group is already being managed by component 5. The other two sub-groups are constructed using components 2, 3, 4, and 6. Two of these are selected to be the new sub-group supervisors (2 and 5), while the other two become the supervised components of the new sub-groups (3 and 6). In the tuple space the middleware creates two new coordination tuples for the groups managed by components 2 and 5, and updates the application group and coordination tuples that describe the application and coordination groups managed by component 1. When the middleware divides a group it takes all the group's components, including the supervisor, and uses them to create n groups that are placed within the group topology. In the bottom example of Figure 4, the middleware divides group B into two new groups supervised by components 2 and 5 respectively. Component 4 is supervised in the group managed by component 2, while component 6 is supervised in the group managed by component 5. In the tuple space the middleware deletes the coordination tuple that describes group B, and creates new tuples for the groups managed by components 2 and 5. It also updates the coordination tuple that described the group managed by component 1 accordingly.

5 A Self-coordinating Greenhouse

The most important factors for the quality and productivity of plant growth are temperature, humidity, light, and the level of carbon dioxide. Continuous monitoring of these environmental variables can help us be more aware of the plants' needs, as well as speed up their growth while achieving remarkable energy savings, especially during the winter. The problem is becoming more and more complex because of the size of some (industrialized) greenhouses and because of the variety of plans that need to co-exist one by the other. These factors require means to improve efficiency, and make local adjustments to the lights, ventilation, irrigation, heating and the other systems supporting the greenhouse.

In our example the greenhouse has been divided into different areas. Each section has sensors to monitor the area's temperature, humidity, and CO_2 level, as well as actuators to control fans, the irrigation system, and fertilization sprinklers. Plants are brought in and out of the greenhouse on carts, which can be of different sizes and contain many plants; they must however all have the same needs in terms of temperature, humidity, and CO_2 level. In practice, the most straight-forward strategy is to keep plants of the same kind, and of the same age, on the same cart. Our goal is to configure the greenhouse to let carts with similar needs —in terms of temperature, humidity, and CO_2 levels— to be placed in the same grid area. The continuous monitoring of vital values, along with the proper activation of the fans and irrigation systems, help maintain the appropriate micro-climates for the different species.

We modeled the greenhouse by introducing a group for each grid area in the greenhouse, one component for each sensor, and one component for each actuator. We then introduced one component for each of the carts that can enter the greenhouse. Finally, each grid area was given a supervisor component (a dedicated server) capable of identifying the needs of its carts, collecting monitoring data from its sensors, and issuing enactment directives to the actuators. The model of the greenhouse is completed by the addition of a group that re-unites all the area-level supervisors of the greenhouse so that they can cooperate. Figure 5 shows the initial configuration of two of the greenhouse's grid areas, in

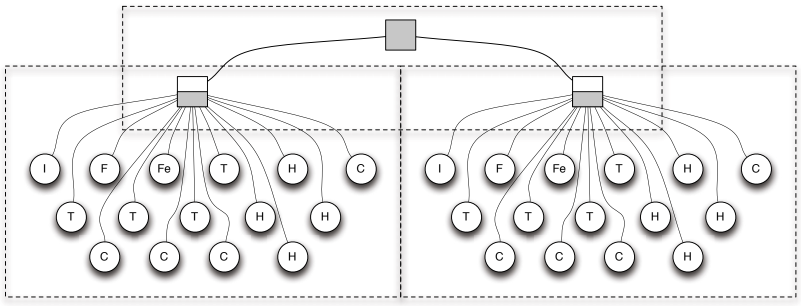


Fig. 5. Areas represented as groups

which there are no carts. The area's supervisor is shown as a square. A circle with an "I" stands for the irrigation actuator, a circle with an "F" stands for the fan, and a circle with an "Fe" stands for the fertilization sprinkler. Circles with a "T" are temperature sensors, circles with an "H" are humidity sensors, and circles with a "C" are CO_2 level sensors. Although the figure only shows four sensors for each type, the actual system actually had many more sensors.

The greenhouse uses A3-TAG to gracefully re-configure itself, both at the application and coordination level, to oversee its operation, and cope with anomalies. The following examples cover five basic scenarios about possible anomalies.

5.1 Sudden Change in a Micro-climate

The first scenario we tackle is the sudden rise or fall, within an area, of its temperature, humidity, or CO_2 level. Very rapid changes in a micro-climate can be very bad for the plants, therefore the area's supervisor needs to react as soon as possible.

Let us consider the case in which the temperature of a single area gets too hot. Under normal circumstances sensors are configured to communicate their average readings to the supervisor every four hours, and to send an emergency notification as soon as they sense a temperature that is too high. Since a sensor can malfunction, or be placed in the wrong position, a single emergency notification is not enough to turn on the fans; the supervisor requires that at least ten sensors report the same high temperatures.

If the problem exists, however, many sensors may notify the supervisor in a very short amount of time, and this may cause congestion. Therefore, we decided to implement the following strategy. As soon as a sensor reads a temperature that is too high, instead of communicating it to the supervisor, it creates a new "Temperature Emergency" sub-group, becomes its supervisor, and connects it to the area's supervisor. If other sensors also read temperatures that are too high, instead of telling the supervisor, they join the emergency group. If more than one emergency group is created concurrently, the groups are merged into a single one. The supervisor of the emergency group continuously monitors the number of sensors it has in its group. If this number grows beyond a threshold, an emergency

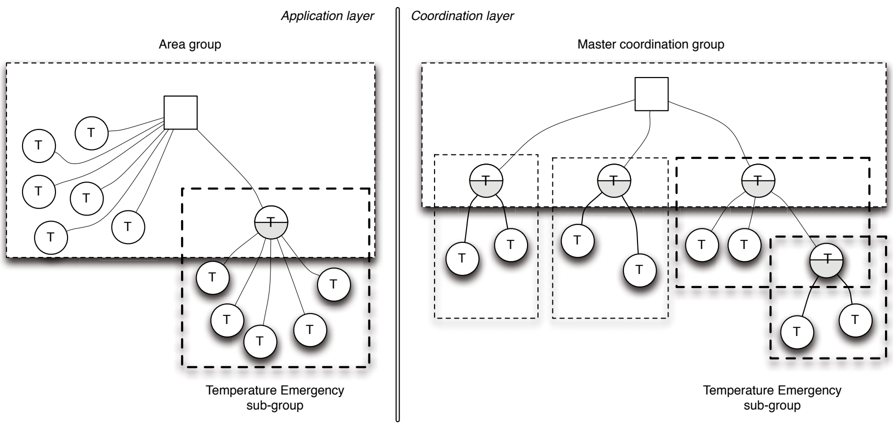

Application layer
Coordination layer
Area group
Master coordination group
T
Temperature Emergency sub-group
Temperature Emergency sub-group

Fig. 6. The Temperature Emergency sub-group

ticket is sent immediately to the supervisor of the area, which then decides to turn on the fans. Figure 6 shows the new configuration in the area's group, as well as what is going on within the middleware in terms of coordination groups. The application layer shows two groups: the area's main group, supervised by a dedicated component, and the temperature emergency group, supervised by a sensor that is also a supervised component in the main group. The coordination layer shows five groups: the master coordination group, two sub-groups for managing the sensors that are not seeing high temperatures, and two groups for the temperature emergency group. With this setup, if the sensor supervising the emergency group fails, the middleware automatically chooses a substitute sensor from within the group and promotes it be the new supervisor. This has the effect of reconnecting the emergency group to the area's coordination group.

5.2 Grid Area Server Dies

The second scenario we tackle in this chapter is the sudden failure of an area's supervisor (server), and how to cope with the emergency while the greenhouse staff replaces it. When a server fails, an entire area becomes unmanaged, and this can have strong negative consequences on its micro-climate. Our solution is to exploit the group that re-unites all the servers to elect a replacing server, that is, the server of one of the other areas, that can take over the failing one. The selected server now needs to manage two different areas, with two different sets of requirements in terms of micro-climate.

Figure 7 shows the application-level view of the two adjacent areas. The left area's server has failed, and therefore it is no longer part of the system. The group that re-unites all the servers in the greenhouse selects the server in the right area to take over. The middleware automatically connects all the components that were connected to the failing server to the new server, effectively creating the situation shown in Figure 7. In our application we decided to piggyback sensor

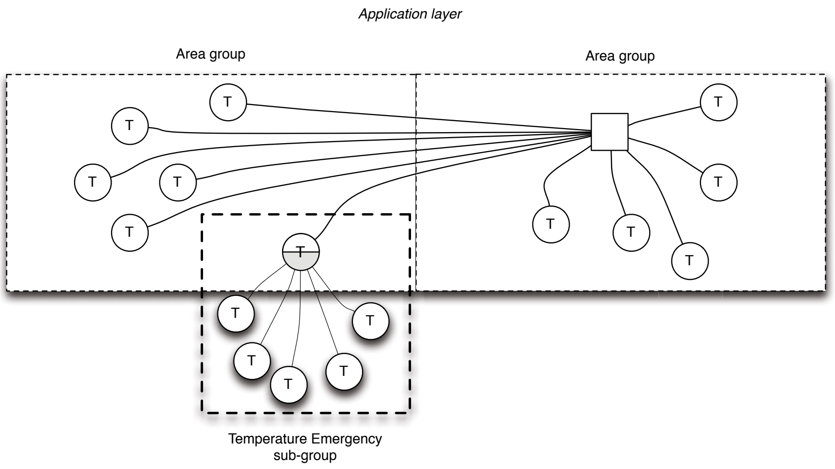


Fig. 7. A grid area server fails

ids and application group ids to all the messages that are sent to a server. This allows servers to effectively manage more than one area at a time.

As soon as all the application and coordination groups reflect this new situation, the server exploits A3-TAG's state management feature to recover the micro-climate requirements of the new group it has to manage. Together with the requirements it also receives information about what chores the old server was entrusted with, and what chores it had already completed before failing. For example, let us imagine that the server was told to start the irrigation system once a night, at midnight, and to keep it on for fifteen minutes. If the server fails before the fifteen minutes are over, the new server needs to know that the irrigation system had started, and at what time, in order to correctly complete the task. Finally, the new server also checks the tuple-space to see if there are any pending messages for the old server that were not correctly delivered. This is particularly important in the case shown in Figure 7, in which the old server was connected to a temperature emergency group that may, or may not, have already issued its emergency notification. Indeed, it is possible that the notification was sent, but that the server failed before receiving it. Thanks to the communication tuples in the tuple-space, this potentially harmful situation is easily managed, and action can be taken immediately, without waiting for the emergency group to issue a second delayed warning.

5.3 Aging Plants

The third scenario we tackle is related to the fact that plants age over time: they grow into new phases of their lives and change requirements in terms of micro-climate. When the plants on a cart change their requirements, the cart communicates this to the area's supervisor through an appropriate message. These messages lead to changes in the desired temperature, humidity, and CO_2

levels, and in the maximum deviations that would be considered acceptable for these values. The reason is that when physical objects, such as carts, share a common space with different requirements, it is impossible to satisfy all of them. Instead, the system needs to resolve conflicting requirements by adopting an appropriate strategy. In our example we have decided to adopt a "Least Misery" strategy, that is one that results in the least discomfort for all the requesting plants when averaged.

When a cart changes its requirements, the server recalculates the averages, and checks them against the requirements it has collected for all the carts it is managing. If all the carts in the area can accept the new averages, the server instructs the area's actuators to implement the change in the micro-climate. If even one cart cannot accept the new averages, the server refuses the cart's new requirements and initiates a strategy that will migrate it to a different area.

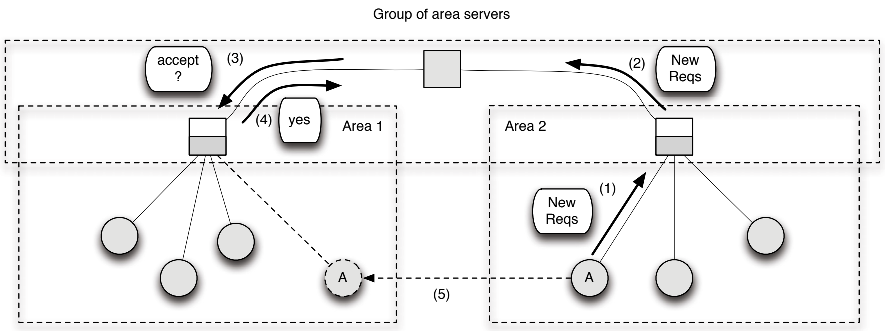


Fig. 8. Finding a new group for a cart of aging plants

In order to migrate the cart to a different area, the server exploits the group that re-unites all the servers in the greenhouse (see Figure 8). In practice, the cart's requirements are sent to the top-level supervisor (step 1), which broadcasts them to all the area servers (step 2). If the areas have enough physical room to accept the cart, they simulate the "least misery" strategy to see what impact it would have on their micro-climate and on the carts they are already managing. A group will declare it can accept the cart only if this does not cause problems to any of its other carts (step 3). If more than one acceptable group is found, the server migrates the cart to the one that is nearest (step 4). If no group is found, the server signals the problem to the greenhouse's administration.

5.4 Sudden Surge in Specific Plant Requests

The fourth scenario is about the sudden surge in requests for a specific kind of plant. This is typically tied to festivities. For example, in certain geographical areas it is common for greenhouses to stock many poinsettias during the month of December. The reason is that it is a flower commonly associated with Christmas.

In this case the greenhouse's goal is to facilitate the selling and re-stock of this flower. For this reason, the greenhouse decides to use an area that is near its entrance, and to extend it with a temporary setup that occupies part of the greenhouse's parking lot, with temporary fans, a temporary irrigation system, and temporary sensors.

Since the greenhouse does not want to install an additional server component just for a few weeks, the system is built to consider the external area as an extension of the internal one. In practice, the system sees one big application-layer group, managed by a single server. The high volume of poinsettias entering and leaving the system provides a good stress test for A3-TAG, since the system needs to increase its internal coordination groups to avoid congestion, and to contract then whenever possible. Indeed, as the topology of the coordination groups expands, more carts need to behave as supervisors and route messages. However, they become failure points that, if lost, can cause the coordination group topology to split into two disconnected sub-topologies. These scenarios are entirely managed by the middleware according to the strategies presented in Section 4.

5.5 Plant Illness Emergency

The fifth and last scenario regards the insurgence of a plant disease that needed to be cured, and confined so that it does not spread to nearby plants. In our greenhouse example there are no sensors that can automatically discover a disease. Instead, human experts visit the premises once a day to check for sick plants. When a sick plant is found, the expert uses a smart portable device to connect to the area being monitored and send a sickness emergency message. Depending on the sickness being signaled, the server can react in different ways. It can decide to react locally by distributing the appropriate medicine through the area's fertilization sprinklers, or it can decide to interact with the other servers in the greenhouse, and have the medicine sprayed in selected neighboring areas or in all the areas in the greenhouse. Whichever the case, the area servers proceed independently and turn on their fertilization sprinklers.

Sprinkling medicine is not the only solution that the area's server can enact. It can also try to contain the sickness by moving the cart with the sick plants to a group that only contains plants that cannot be infected. This would hopefully have the effect of slowing down the contagion, allowing less medicine to be sprinkled in the greenhouse. This re-organization is performed similarly to how we managed the migration of aging plants. Indeed, the server needs to find a group that has enough room to accept the cart with the sick plants, only has plants that cannot be infected, and whose micro-climate will continue to be acceptable for its plants even if the new cart is accepted with its requirements. Finally, depending on the sickness and on the state of advancement of the contagion, the system can also decide to remove the cart from the system entirely. In this case, the system sends a notification to the greenhouse's administrative team.

6 Evaluation

The greenhouse scenario requires numerous wireless sensors to be deployed into a relatively small area, in order to effectively monitor the temperature. When networks become crowded many typical wireless networking problems can be aggravated. For example, we can have node interference, routing proliferation, and nodes can use high transmission power to exchange messages directly with distant nodes, limiting the reuse of the wireless bandwidth.

A typical solution is to limit the number of neighbors that are within range of one another. However, this must be achieved in such a way that no messages are ever lost, not even when the network is showered with events. A3-TAG's topology control helps overcome this problem. Instead of using the network to the full extent of its connectivity, we choose to artificially restrict the network's topology. The topology is determined by the subset of active nodes and links along which direct communication is possible. In A3-TAG we have a network in which a number of nodes play the special supervisor role. These nodes constitute the network's communication "backbone". Indeed, we communicate using the links within this backbone, and direct links from all the other nodes towards the backbone. Furthermore, our self-adaptive topology tries to adapt itself to the needs of the ongoing communication, even when the network is showered with events.

To evaluate our approach we need to assess its "scalability", its "algorithm overhead", that is the number of additional messages needed to ensure the desired topology control, its "robustness to churn", that is its capability to adapt in acceptable time when there is a high node churn rate, and its "throughput effectiveness", that is its capability to optimize the active links without jeopardizing the network's connectivity and without causing message loss.

Although we have fully implemented the greenhouse example in a simulated environment, to assess A3-TAG's performance with respect to the above metrics, it is sufficient to analyze the behavior of a single, yet highly crowded, area of the greenhouse. In our tests we setup an application group for a group of sensors that autonomously read temperature values and send them to their supervisor once every four hours. When the temperatures become too high or too low, all the sensors attempt to notify the supervisor at the same time, creating high traffic. In our experiments we dynamically changed the number of components in the group to analyze how the middleware scaled, what its algorithm message overhead was, and its degree of throughput effectiveness. We also artificially created different churns of components, with different rates, to analyze its robustness.

The tests were conducted under the following assumptions. First, we assume that the setup does not introduce any sort of communication limitations. Second, we assume that we can have up to 60 messages per second in our network (MAC 55.6Kbps, payload 40Kbps). Trying to send more messages will cause message loss and bandwidth reduction. Therefore, throughput effectiveness is achieved when the number of messages per second sent to a single node is less than 60.

The evaluations were performed as fixed-frequency cycle-based simulations in PeerSim [10]. In PeerSim peers are activated in a sequential fashion, and are executed in isolation, without concurrency. This cycle-based model allowed us to scale the experiments up to 10^4 peers. The peers were randomly scattered within a 7-unit square, and the supervisor node was located in the center of the square. For each experiment we performed 32 independent runs, obtaining an estimated P-value of less than $0.5 * 10^{-1}$, which means that the experiments are statistically significant. Here are some of our conclusions:

– **Scalability:** Our experiments have shown that we are able to apply topology control up to 10^4 nodes, and that the approach is conceptually independent of the network size. In our experiment we established the number of PeerSim cycles required to go from a completely unbalanced network to an optimized A3-TAG topology of coordination groups. When we changed the size of the network from 100 to 5000 nodes, the number of PeerSim cycles required to adapt the system increased from 6 cycles to 8. The expected worst-case time complexity for topology control is therefore $O(1)$. For more details on this conclusion the reader can look to [7].
– **Algorithm Message Overhead:** Our experiments have shown that the worst-case hotspot message complexity per peer is $O(log(N))$. Hotspot message complexity is defined as the maximum number of messages sent, received, and forwarded by one single element in the network. In a distributed system the communication load among all elements needs to be balanced to minimize the hotspot message complexity and avoid overloading any element. To measure this metric we had each component send a network-wide broadcast.
– **Robustness to Churn:** In our experiments we fixed the network size and started randomly killing components, and then pinged all the nodes after 4000 milliseconds. We further extended the experiment by re-inserting the components into the system and waiting another 4000 milliseconds to ping them again. All the components were able to respond to the ping correctly, regardless of the churn rate. This experiment is similar to one discussed in [7], where we compared our approach with a completely decentralized solution based on LIME.
– **Throughput Effectiveness:** Figure 9 shows how A3-TAG compares to a flat network. The graph shows the average number of messages received by each node per cycle. With A3-TAG this number is shown to be always less than 40. While, in the flat network, when we go higher than 300 nodes the average grows to more than 60 messages. The supervisor node might end up receiving a message several times (from different directions), while other messages might get lost.

Our evaluations have still not allowed us to understand how to set the optimum threshold for nodes in a single coordination group. To gather an initial assessment, we followed the guidelines proposed in [9], which state that the number of neighbors of a single node in a balanced network should grow like $a.log(N)$, where N is the total number of nodes and a is a bound constant. For a less than 0.074 the network is asymptotically disconnected, while for a greater than 5.1774 the network is asymptotically connected.

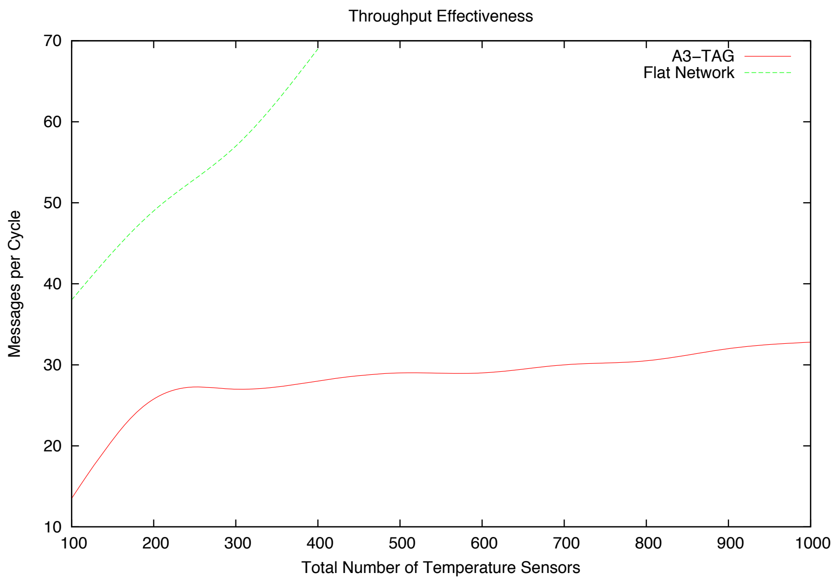


Fig. 9. Throughput Effectiveness

In our experiments we placed A3-TAG's coordination group membership threshold to a maximum of 4 nodes, when we had up to 50 nodes in the network. We then altered this number according to $a * log(N)$, with a equal to 9.5. This allowed us to obtain a fair comparison with flat networking, and allowed us to avoid having un-optimized topologies. For example, if we were to fix the threshold to 5 when we have 100 nodes, and then keep the same threshold and increase the number of nodes to 5000, the number of coordination groups would be too high, and the grouping would be ineffective. If we were to set the threshold to 40 nodes when we have 5000 nodes in the network, and then decrease the number of nodes to 100, we would end up with just 3 coordination groups. This would not be much better than having a single big group for all the components. Although this solution allowed us to gather quite good results, further investigation is necessary.

7 Conclusions and Future Work

Felicitous systems call for suitable self-adapting middleware infrastructures and programming models to deal with large, dynamic software systems. A3-TAG exploits the group abstraction to provide designers with powerful means to tackle the design and operation of these systems. Moreover, thanks to the A3-TAG middleware, application components do not need to explicitly deal with group management, or be worried with inter-component communication. Each node automatically inherits these capabilities from the group(s) it belongs to. A3-TAG also continuously adapts its own internal component connection topology to preserve group integrity and to ensure message delivery is performant and robust.

A3-TAG has been developed to avoid any single point of failure and bottlenecks. Awareness lets failing supervisors be easily replaced, and state management lets the new supervisors immediately start from where the old ones left off. Since the number of supervisors and groups can be changed dynamically, bottlenecks can be foreseen and prevented accordingly. All these features were demonstrated in the context of a self-adaptive greenhouse.

The current implementation of A3-TAG is quite demanding in terms of resources and computing capabilities, but many of the underlying concepts could be used also within networks of devices with limited capabilities (e.g., sensor networks). To this end, we are thinking of both a *light* and *hybrid* versions of A3-TAG. The former aims to address devices with limited capabilities, while the latter outsources the supervision to the cloud and concentrates on the integration of heterogeneous resources.

References

1. Mainwaring, A., Culler, D., Polastre, J., Szewezyk, R., Aderson, J.: Wireless Sensor Networks For Habitat Monitoring. In: Proceedings of the 1st annual ACM International Workshop on Wireless Sensor Networks and Applications, WSNA 2002, pp. 88–97 (2002)
2. Juang, P., Oki, H., Wang, Y., Martonosi, M., Peh, L., Rubenstein, D.: Hidden vs. Exposed Terminal Problem in ad Hoc Networks. In: Proceedings of the Australian Telecommunication Networks and Applications Conference, ATNAC 2004 (2004)
3. Burrell, J., Brooke, T., Beckwith, R.: Vineyard Computing: Sensor Networks in Agricultural Production. In: Proceedings of the IEEE Pervasive Computing, vol. 3, pp. 38–45 (2004)
4. Petriu, E.M., Georganas, N., Petriu, D.C., Makrakis, D., Groza, V.: Sensor-Based Information Appliances. IEEE Instrumentation and Measurement Magazine 3, 31–35 (2000)
5. Deshpande, A., Guestrin, C., Madden, S.R., Makrakis, D., Groza, V.: Resource-Aware Wireless Sensor-Actuator Networks. IEEE Data Engineering 28 (2005)
6. Al-Ars, Z., Kootkar, S.: Design and Implementation of Reliable Wireless Sensor Networks A Case Study in Commuter Trains. In: Proceedings of the Workshop of Program for Research on Integrated Systems and Circuits, ProRISC 2007, pp. 303–306 (2007)
7. Guinea, S., Saeedi, P.: Coordination of Distributed Systems through Self-Organizing Group Topologies. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS, pp. 63–72 (2012)
8. Ali, M., Bohm, A., Jonsson, M.: Wireless Sensor Networks for Surveillance Applications - A Comparative Survey of MAC Protocols. In: Proceedings of the 4th International Conference on Wireless and Mobile Communications ICWM, ICWMC 2008, pp. 399–403 (2008)
9. Blough, D.M., Leoncini, M., Resta, G., Santi, P.: The K-Neigh Protocol for Symmetric Topology Control in Ad Hoc Networks. In: Proceedings of the 4th ACM International Symposium on Mobile ad Hoc Networking & Computing, MobiHoc 2003, pp. 141–152 (2003)

10. Jelasity, M., Montresor, A., Jesi, G.P., Voulgaris, S.: The Peersim Simulator, `http://peersim.sf.net`
11. Perbellini, G.: A Middleware-centric Design Methodology for Networked Embedded Systems. Ph.D. Thesis, Universita' degli Studi di Verona a Dipartimento di Informatica, Italy (2009)
12. Costa, P., Coulson, G., Gold, R., Lad, M., Mascolo, C., Monttola, L., Picco, G.P., Sivaharan, T., Weerasinghe, N., Zachariadis, S.: The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario. In: Proceedings of the 5th Annual International Conference on Pervasive Communications, PerCom 2007, pp. 69–78 (2007)
13. Felicitous Computing Institute, `http://fci.comp.nus.edu.sg/`
14. Baresi, L., Guinea, S.: A-3: an Architectural Style for Coordinating Distributed Components. In: Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture, WICSA 2011, Colorado (2011)
15. Baresi, L., Guinea: A-3: Self-Adaptation Capabilities through Groups and Coordination. In: Proceedings of the 4th India Software Engineering Conference, ISEC 2011, India, pp. 11–20 (2011)
16. Román, M., Islam, N.: Dynamically Programmable and Reconfigurable Middleware Services. In: Jacobsen, H.-A. (ed.) Middleware 2004. LNCS, vol. 3231, pp. 372–396. Springer, Heidelberg (2004)
17. Ingram, D.: Reconfigurable Middleware for High Availability Sensor Systems. In: Proceedings of the International Conference on Feature Interactions in Telecommunications and Software Systems, pp. 12–30 (2005)
18. Fidler, E., Jacobsen, H.A., Li, G., Mankovski, S.: The PADRES distributed publish/subscribe system. In: Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems, DEBS 2009, vol. 20, pp. 1–11 (2009)
19. 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.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 27–47. Springer, Heidelberg (2009)
20. D-Bus home page, `http://dbus.freedesktop.org/`
21. Mamei, M., Zambonelli, F.: Programming Pervasive and Mobile Computing Applications with the TOTA Middleware. J. Pervasive and Mobile Computing 18, 1–51 (2004)
22. Schuhmann, S., Herrmann, K., Rothermel, K.: Efficient Resource-Aware Hybrid Configuration of Distributed Pervasive Applications. In: Proceedings of the 8th International Conference on Pervasive Computing, Hilsinki, pp. 373–390 (2010)
23. Kolbeck, B., Hgqvist, M., Stender, J., Hupfeld, F.: Fault-Tolerant and Decentralized Lease Coordination in Distributed Systems. Technical Report, Zuse Institute Berlin (2010)
24. Ceriotti, M., Murphy, A.L., Picco, G.P.: Data Sharing vs. Message Passing: Synergy or Incompatibility? An Implementation-Driven Case Study. In: Proceedings of the 23rd Symposium on Applied Computing, SAC 2008, pp.100–107 (2008)
25. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and Evaluation of a Wide-Area Event Notification Service. J. ACM Transactions on Computer Systems 19, 332–383 (2001)
26. LIME home page, `http://lime.sourceforge.net/index.html`
27. rtag: an extension to A-3 home page, `http://code.google.com/p/rtag/`
28. REDS (REconfigurable Dispatching System), `http://zeus.ws.dei.polimi.it/reds/`

29. JGroups home page, `http://www.jgroups.org`
30. Gelernter, D.: Generative communication in Linda. J. ACM Transactions on Programming Language and Systems 7, 80–112 (1985)
31. TSpaces home page, `http://www.almaden.ibm.com/cs/TSpaces/`
32. LighTS home page, `http://lights.sourceforge.net/`
33. Viroli, M., Casadei, M., Montagna, S., Zambonelli, F.: Spatial Coordination of Pervasive Services through Chemical-Inspired Tuple Spaces. J. ACM Transactions on Autonomous and Adaptive Systems, TAAS 6, 1–24 (2011)
34. Fok, C.L., Roman, G.C., Lu, C.: Enhanced Coordination in Sensor Networks through Flexible Service Provisioning. Journal of Field and V. T. Vasconcelos (2009)
35. Roman, M., Hess, C.K., Cerqueira, R., Ranganathan, A., Campbell, R.H., Nahrstaedt, K.: Gaia: A Middleware Infrastructure to Enable Active Spaces. In: Proceedings of the IEEE Pervasive Computing, vol. 1 (2002)
36. Valetto, G., Snyder, P., Dubois, D., Di Nitto, E., Calcavecchia, N.: A Self-organized Load-balancing Algorithm for Overlay-based Decentralized Service Networks. In: Proceedings of the IEEE International Conference on Autonomic Computing (ICAC), pp. 168–177 (2011)
37. Edward, G., Garcia, J., Tajalli, H., Poescu, D., Medvidovic, N., Sukhatme, G.: Architecture-driven Self-adaptation and Self-management in Robotic Systems. In: Proceedings of Software Engineering for Self-Adaptive Systems, SEAMS, pp. 142–151 (2009)
38. Roth, M., Schmitt, R., Kiefhaber, R., Kluge, F., Ungerer, T.: Organic Computing Middleware for Ubiquitous Environments. In: Organic Computing - A Paradigm Shift for Complex Systems, vol. 1, pp. 339–351. Springer, Basel (2011)
39. von Renteln, A., Brinkschulte, U.: Implementing and Evaluating the AHS Organic Middleware - A Firt Approach. In: Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC, pp. 163–169 (2010)
40. Pacher, M., Brinkschulte, U.: Implementation and Evaluation of a Self-organizing Artificial Hormone System to Assign Time-dependent Tasks. In: Concurrency and Computation: Practice and Experience. John Wiley & Sons (2011)
41. Weyns, D., Holvoet, T.: An Architectural Strategy for Self-Adapting Systems. In: Proceedings of the 2nd International Workshop on Software Engineering for Adaptive and Self-Managing Systems (2007)
42. Holvoet, T., Weyns, D., Valckenaers, P.: Patterns of Delegate MAS. In: Proceedings of the 3rd IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO, pp. 1–9 (2009)
43. Shehory, O.M., Sycara, K., Jha, S.: Multi-Agent Coordination through Coalition Formation. In: Intelligent Agents IV Agent Theories, Architectures, and Languages, pp. 143–154. Springer, Berlin (1998)
44. Montresor, A.: A Robust Protocol for Building Superpeer Overlay Topologies. Technical Report UBLCS-2004-8 (2004)
45. Milo, T., Zur, T., Verbin, E.: Boosting Topic-based Publish-subscribe Systems with Dynamic Clustering. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 749–760 (2007)
46. Caviglione, L., Giuseppe, C., Gianuzzi, V.: Architecture of a Communication Middleware for VANET Applications. In: Proceedings of the 10th IFIP Annual Mediterranean Ad Hoc Networking Workshop, pp. 111–114 (2011)

47. Efthymiopoulos, N., Christakidis, A., Denazis, S., Koufopavlou, O.: L-CAN: Locality Aware Structured Overlay for P2P Live Streaming. In: Pavlou, G., Ahmed, T., Dagiuklas, T. (eds.) MMNS 2008. LNCS, vol. 5274, pp. 77–90. Springer, Heidelberg (2008)
48. Wang, Y., Nakao, A.: On Cooperative and Efficient Overlay Network Evolution Based on a Group Selection Pattern. IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics - Special Issue on Game Theory Archive 40(3), 493–504 (2010)
49. Rene Meier, V.C.: Steam: Event-based Middleware for Wireless Ad Hoc Networks. In: Proceedings of the Interational Workshop on Distributed Event-Based Systems, Austria (2002)
50. Jelasity, M., Babaoglu, O.: T-Man: Gossip-based Overlay Topology Managemen. In: Proceedings of the 3rd International Workshop on Engineering Self-Organizing Applications, pp. 1–15 (2005)

Accurate Proactive Adaptation of Service-Oriented Systems

Andreas Metzger, Osama Sammodi, and Klaus Pohl

Paluno (The Ruhr Institute for Software Technology)
University of Duisburg-Essen, Essen, Germany
{andreas.metzger,osama.sammodi,klaus.pohl}@paluno.uni-due.de

Abstract. As service-oriented systems are increasingly composed of third-party services accessible over the Internet, self-adaptation capabilities promise to make these systems become robust and resilient against third-party service failures that may negatively impact on system quality. In such a setting, proactive adaptation capabilities will provide significant benefits by predicting pending service failures and mitigating their negative impact on system quality. Proactive adaptation requires accurate quality prediction techniques; firstly, because executing unnecessary proactive adaptations (due to false positive predictions) might lead to additional costs or follow-up-failures; secondly, because proactive adaptation opportunities may be missed (due to false negative predictions). This book chapter reviews solutions for measuring and ensuring the accuracy of online service quality predictions. It critically analyses their applicability in the setting of third-party services and supports this analysis with empirical data.

Keywords: Service-oriented computing, software services, accuracy, metrics, online quality prediction, online failure prediction.

1 Motivation

Service-orientation is increasingly adopted as a paradigm for building highly dynamic, distributed software systems. Those service-oriented systems are built by composing and integrating individual software services. As a consequence and in stark contrast to “traditional” software components, not only the development, quality assurance, and maintenance of the software can be under the control of third-parties, but the software itself can also be executed and managed by third-parties [46].

There is an evident trend that service-oriented systems will increasingly be composed of *third-party services* accessible over the Internet [56,1]. These third-party services may include infrastructure services (such as compute and storage resources), platform services (such as middleware and commodity services), software services (offering specific business functionality), as well as complete applications offered through the Software-as-a-Service delivery model [5]. As a consequence, the capabilities and quality of service-oriented systems will more and more depend on the quality of their third-party services. Specifically, this means that service-oriented systems have to become robust and resilient against failures of their third-party services.

We consider self-adaptation capabilities a key solution to ensure robustness and resilience against third-party service failures [46,13,41]. In this setting, *proactive* adaptation capabilities promise significant benefits over reactive adaptation capabilities. Proactive adaptation allows service-oriented systems to respond to imminent failures, thus preventing their actual occurrence or mitigating their impact. As a result, proactive adaptation may avoid the need for costly repair or compensation activities [26,45,50,6,33,34,28,60].

Online quality prediction techniques are employed to anticipate imminent failures. Extrapolating past observations of service quality (typically collected by monitoring the service execution [47]), those techniques provide short-term predictions of failures, which in turn may trigger proactive adaptations during the run-time of the service-oriented system.

1.1 Problem Statement and Contributions

The goal of quality prediction is to forecast future quality accurately. Informally, this means that quality prediction should forecast as many failures as possible, while – at the same time – generating as few false "alarms" as possible [49]. Accurate predictions are important for proactive adaptation, such as to avoid the execution of unnecessary proactive adaptations, as well as to not miss proactive adaptation opportunities.

This book chapter focuses on the following two important challenges for what concerns accurate quality predictions for adaptive service-oriented systems:

Measuring Accuracy: The literature provides a wide range of metrics to assess the accuracy of prediction techniques. Those metrics have been proposed for domains such as software engineering (defect prediction), fault-tolerant systems, cluster computing, and business process management. They include prediction error metrics (such as root mean squared error, mean absolute error, and relative error), as well as contingency table metrics (such as precision, recall and specificity). Although the existing metrics may work well for the above domains, it remains open whether they can be applied and provide useful results in the domain of service-oriented systems.

As one key contribution, this chapter revisits existing metrics and analyzes their potential limitations when applied to service-oriented systems. On the one hand, we observe that prediction error metrics may be of limited use in assessing how accurately the need for adaptation may be predicted. On the other hand, our analysis shows that contingency metrics may be very sensitive to parameters (such as thresholds for QoS values) and thus may provide quite different results in slightly different contexts.

Ensuring Accuracy: To ensure accurate predictions, various prediction techniques and prediction models for different domains have been proposed in the literature (including time series predictions, such as moving average and exponential smoothing). Due to the fact that online quality predictions aim at performing short-term predictions, there is a certain conflict between being responsive enough to changes in past observations and being too responsive and thus being perturbed by noise and fluctuations in the past observations [12].

As a key contribution, the chapter shows that similar problems are faced for short-term prediction of service quality. Complementing previous suggestions on improving the prediction models [12,3,2], we propose collecting additional (more frequent) data points to improve prediction accuracy. Specifically, we show that online testing for service-oriented systems allows systematically collecting such additional observations.

1.2 Chapter Structure

After an introduction to the fundamentals and state of the art of online quality prediction for service-oriented systems in Section 2, we review metrics for accuracy assessment in Section 3 and techniques to ensure prediction accuracy in Section 4. We conclude by sketching remaining research challenges in Section 5.

2 Fundamentals and State of the Art

This section first provides a brief introduction to adaptive service-oriented systems (Section 2.1), followed by an overview of the state of the art of online quality prediction (Section 2.2). The section then elaborates on the need for accurate proactive adaptation (Section 2.3) to motivate the contributions in the remainder of the chapter. Finally, it describes the setup of experiments (Section 2.4) that we will use to empirically support our findings and discussions. Thus, this section provides the fundamentals for what follows in Sections 3 and 4.

2.1 Adaptive Service-Oriented Systems

The *Service Oriented Architecture* (SOA) constitutes a set of guiding principles [32] for building service-oriented systems. Thanks to these principles, services may separate ownership, maintenance and operation from the use of the software. Service users thus do not need to acquire, deploy, and run software because they can access its functionality remotely through service interfaces. Ownership, maintenance, and operation of the software remains with the service provider [46].

As a result, we can observe the proliferation of *third-party software services* that enable organizations to flexibly outsource business functions (typically commodity functions) and to focus on the innovative functions which differentiate one organization from another. As an example, at the time of writing, the seekda Web service search engine already lists more than 28,000 services on its web-site[1].

Integrating third-party services into service-oriented systems implies that these systems will be subject to changes at run-time in their execution environment that

[1] `http://webservices.seekda.com/`

are only under limited control by service integrators [58,55]. Examples include service changes, such as new versions of services that are incompatible with previous versions, the discontinuation of service offerings by their providers, as well as fluctuations in service quality, such as performance, availability and reliability.

Over the past years, many efforts have been made towards adaptive service-oriented systems. *Adaptation* refers to the ability of a system to dynamically modify its behavior and/or structure in response to its perception of the environment and the system itself, as well as its requirements [15,35]. Adaptive capabilities become essential features to guarantee robust and resilient system operation, once dynamic changes are not exceptions but the normal behavior imposed on these systems.

Extensive surveys on adaptive service-oriented systems are provided in [47] and [38]. The adaptation capabilities that are introduced in the literature fall into the following two major clusters:

Reactive Adaptation refers to the case in which the system is modified in response to deviations in system quality, i.e., failures that are actually observed by the users of the system. Repair and/or compensation activities have to be executed as part of the adaptation in order to mitigate the effects of those failures; e.g., the user is paid a compensation, or certain service invocations are rolled back. Besides leading to additional costs due to such compensations, reactive adaptation may have a severe impact on how agilely a system can respond to changes [43,26]. As examples, the execution of reactive adaptation activities on the running system can considerably increase execution time and therefore reduce the overall performance of the running system, or an adaptation of the system might not be possible at all, e.g., because the system has already terminated in an inconsistent state.

Proactive Adaptation refers to the case in which the need for adaptation is anticipated and thus preventive action can be taken such as to avoid failures. Proactive adaptation is thus based on "short-term" predictions, i.e., forecasting imminent failures that require an adaptation of the running system.

Proactive adaptations can be further classified as described below (Figure 1 provides an illustration using a simple service composition):

a. One class of proactive adaptations aims to execute countermeasures such as to compensate the impact of *actual* service failures before they negatively impact on system quality: As a simple example, if the third-party service $s2$ in Figure 1a is responding too slow, online quality prediction is used to determine whether this failure in turn implies that the service-oriented system may respond too slow, and thus may be negatively perceived by its users, or even violate an SLA agreed upon with its users. As a countermeasure, the system could replace the not yet invoked service $s5$ with the faster service $s5'$, thereby compensating for the slow response of service $s2$. Please note that, due to the time delay between the detection of the service failure of $s2$ and the observation of the external failure by the user, there is more flexibility in modifying the system than in the reactive case, i.e., when the system quality has already been violated.

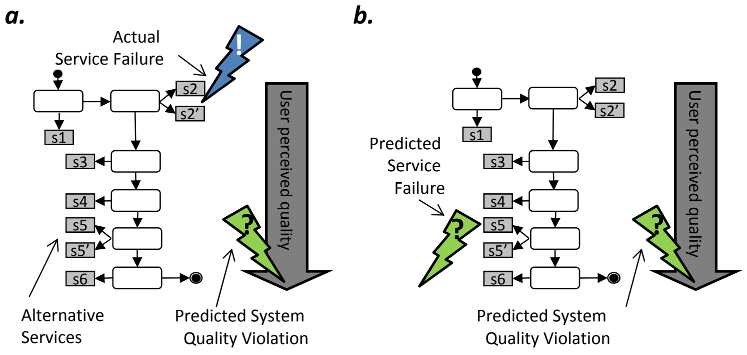


Fig. 1. Adaptation types enabled by online failure prediction: *a.* based on predicted system failure and *b.* based on predicted service failure

b. Another class of proactive adaptations aims to execute activities such as to avoid the impact of *predicted* service failures. Such type of proactive adaptation, allows modifying the system even before a faulty service is actually being executed. If the system is able to predict that a service failure is imminent (but did not yet occur) and that this failure may impact on service quality, the system can be modified before execution reaches the faulty service. As an example, if service $s5$ in Figure 1b is predicted to respond too slow (and thus leading to system performance that is too low), that service could be replaced by a service $s5'$ known to be faster.

2.2 Online Quality Prediction

As indicated above, proactive adaptation requires prediction of service and/or system failures.

In various areas of computer science and software engineering, such online failure prediction or quality prediction has received considerable attention. A recent survey by Salfner et al. provides an excellent overview and taxonomy of the state of the art in the more traditional area of computer-based systems [49]. Yet, compared with the increasing complexity, dynamics, and flexibility in these more traditional areas [49], service-oriented systems face unprecedented levels of dynamism, together with a lack of control over third-party services (see Section 2.1). Thus, different classes of novel techniques as well as adaptations of existing techniques have emerged for predicting the quality of services and service-oriented systems [42].

Below, we list the major types of those techniques proposed for predicting the quality of service-oriented systems and for predicting the quality of individual services, respectively.

2.2.1 Quality Prediction for Service-Oriented Systems

This first class of techniques includes approaches to determine the (end-to-end) quality of a service-oriented system as perceived by its users.

- *Machine Learning/Data Mining* [33,18]: This class of approaches leverages data mining and machine learning capabilities to train prediction models using historic monitoring data. Proposed solutions include variants of statistical methods (such as regression), as well as multi-layer artificial neural networks [25] for quantitative QoS and decision trees [48] for qualitative QoS.
- *Run-time Verification* [20,52]: Run-time verification is a formal analysis technique used to ascertain whether some predefined properties are met at run-time. The proposed solutions typically suggest using run-time model checking of the service composition model to determine whether it will effectively be possible for the execution of the service-oriented system to finish successfully.
- *Static Analysis* [29]: Static analysis systematically examines an artifact to infer certain properties. These properties can include approximations of the future of the computation; e.g., it can be used to predict ranges (i.e., upper and lower bounds) for the behavior of the remainder of the computation. Proposed solutions use the service composition structure of the service-oriented system as a basis to forecast QoS deviations by mapping it to a constraint satisfaction problem (CSP) [4].
- *Simulation* [30,31]: During simulation, dynamic models are executed to mimic the behavior of service-oriented systems and thus to predict their future behavior and quality properties. For example, service compositions are transformed into dynamic models which also model the resources available to execute system (e.g., number of simultaneous threads). These approaches often resort to discrete event simulation tools [51].

2.2.2 Quality Prediction for Individual Services

This second class of prediction techniques aims at forecasting the quality of individual services.

- *Time Series Prediction* [2,14,39]: Time series prediction models employ monitoring data (i.e., past observations of service behavior) to extrapolate the future quality of a service. So far these models have mainly been proposed for response time predictions of service-oriented systems. Typically, time series predictors include models such as windowed means (moving average) or exponential smoothing [49].
- *Predictive Event-Processing* [60,19,44]: Complex event processing (CEP) aims to detect complex events in streams of incoming raw events. CEP enables immediate and automatic response to a set of predefined situations, each characterized up-front during system design and deployment. Predictive event-processing has been proposed as part of *proactive event-driven computing*, a paradigm that combines event prediction capabilities with decision making capabilities, targeted at mitigating the effect of predicting undesired events.
- *Online Testing* [8,22,50,17]: The previous online failure prediction techniques all rely on monitoring mechanisms to collect QoS data. In contrast, quality prediction techniques based on online testing complement monitoring data

with data actively collected by testing. Online testing means that the service-oriented system is tested (i.e, fed with dedicated test input) in parallel to its normal use and operation [9,11].

To keep the remainder of this paper focused, we will limit our discussions to *service* quality prediction approaches, as discussed in this sub-section. Ensuring the accuracy of those predictions will have strong leverage effects on achieving accurate predictions of system quality. Ultimately, online service quality prediction will enable earlier response to imminent issues and thus leaves more opportunities for addressing those issues.

2.3 Need for Accurate Online Quality Prediction

We have seen above that, in order to trigger the proactive adaptation of a service-oriented system, the system needs to be able to predict pending failures. As motivated in the introduction to this chapter, a key goal of quality prediction is to forecast future quality accurately, meaning that as many failures as possible should be forecasted, while generating as few false "alarms" as possible [49].

Below we will elaborate on the relevance of accurate predictions for proactive adaptation. To this end, Figure 2 illustrates the four situations that may occur when performing online service quality prediction.

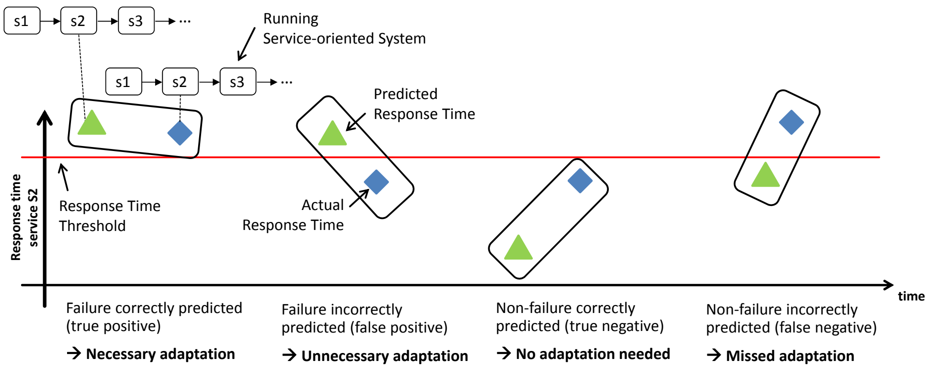


Fig. 2. Four different situations during quality prediction for service $s2$

It should be noted that, assuming that respective monitoring mechanisms are in place, any time a service is executed within the running service-oriented system, its QoS response is observed. These observations in turn provide the input for quality prediction. The diagram illustrates this on the left hand side by sketching the sequence of service along the execution of the service-oriented system. Those service invocations represent the points in time when monitoring is performed and thus QoS can be observed.

As the diagram illustrates, there are two “critical” situations that may occur:

– **Unnecessary adaptations:** False positive predictions may trigger the adaptation of the service-oriented system although the service would have actually worked as expected. Such unnecessary (or unrequired [2]) adaptations can have the following severe shortcomings: Firstly, unnecessary adaptations can be costly. For instance, additional activities such as Service Level Agreement (SLA) negotiation for the alternative services might have to be performed, or the adaptation can lead to a more costly operation of the service-oriented system, e.g., if a seemingly unreliable but cheap service is replaced by a more costly one. Secondly, unnecessary adaptations could be faulty (e.g., if the new service has bugs), leading to severe problems as a consequence. Thirdly, as executing the adaptation takes time, this means that in the worst case, an unnecessary adaptation will leave less time to address actual failures.
– **Missed adaptations**: False negative predictions will not trigger an adaptation, although the service will actually fail and this failure could have been proactively compensated. In case an adaptation opportunity is missed due to inaccurate failure predictions this obviously can lead to the same shortcomings as faced in the setting of reactive adaptations, i.e., it can require compensation or costly repair activities. This means, inaccurate predictions would diminish the overall efficiency of proactive adaptation.

As we will see in the remainder of this chapter, providing accurate failure predictions can become very challenging in the setting of service-oriented systems if third-party services are present. The observed quality and functionality of those third-party services can significantly vary between different service invocations. For instance, the performance of a third-party service might depend on the load of the infrastructure at the provider's side or the network latency if services are offered over the Internet. As an example, a failure observed at one point in time (e.g., unavailability of a service because of an overload of the service provider's infrastructure) can disappear at a later point in time (e.g., the same service now responds because of a lower load of the infrastructure).

2.4 Experimental Setup

In the remainder of this chapter, we will refer to experimental data to support our discussions. We briefly recall the setup of our experiments that we performed to gather that data (details are available online^2).

The experiments are based on: (1) the simulation of an example service-oriented system and its associated third-party services, together with (2) a prototypical implementation of the various quality prediction techniques (including time series prediction models and online testing, see Section 2.2.2). We focus on response time as the QoS property to predict.

The service composition has been constructed and is being simulated such as to cover usage rates in the range from 0.01–0.20, online test rates from 0.15–0.60,

^{2} at `http://www.s-cube-network.eu/asas/asas.pdf` as well as service failure rates from 0.15–0.25. Although the service composition is artificial, it involves data from real services. More precisely, we employ a pre-recorded raw data set that has been produced by Cavallo et al. [14]. During simulation, we thus can retrieve response times for actual services for the respective point in (simulation) time.

The service response time data set we used contains up to 2000 data points per service and serves the following two purposes in our experiment: (1) it is exploited as input for the quality prediction techniques (e.g., considered as monitoring data and testing results based on which a prediction is based); (2) it is used to compute the accuracy of predictions (i.e., serves as source for actual values to check predictions against).

To simulate the actual invocation of services, we randomly determine the points in simulation time (i.e., in the range of [1, 2000]) at which the next service invocation will take place by considering the different usage rates. Similarly, to simulate online testing of a service, we use the online testing rates to determine the next point in simulation time when to gather the "test results".

Concerning the way we have simulated the execution of service invocations and online tests, we strove to be as realistic as possible. Yet, we used the QoS data of the services provided by Cavallo et al. to represent both monitoring data and online test execution results, which may pose a threat to validity. Concerning the generalization of the results, so far, we have considered only response time as the QoS property to predict. As part of our ongoing work, we are performing online testing in a live setting based on a prototype implementation of our solutions, as well as considering other QoS properties (e.g., availability and reliability) and more complex prediction models (e.g., ARIMA, Markov Chains). Concerning the repeatability of our experiments, the detailed experimental setup as well as all experimental data and results are available online at: `http://www.s-cube-network.eu/asas/asas.pdf`. The data set used is published in [14].

3 Measuring Prediction Accuracy

In the literature, various metrics have been proposed and used to evaluate the accuracy of prediction techniques. In this section, we review widely used accuracy metrics and discuss their applicability in the setting of service-oriented systems. Specifically, we scrutinize metrics for assessing numeric predictions (such as QoS values) in Sections 3.1 and 3.2, as well as binary predictions (failure / non-failure) in Sections 3.3 and 3.4.

3.1 Assessing Numeric Predictions

Metrics used for assessing numeric predictions quantify the size of the error when predicting numerical values, such as response time or availability. Table 1 summarizes typical metrics from the literature that are used for this purpose.

Mean Squared Error (MSE) is the basic and most-commonly used metric [57]. Root Mean Squared Error ($RMSE$) is useful in that it allows the error to be

Table 1. Metrics for measuring numeric predictions based on [57]. $\widehat{m}_i$ predicted, a_i actual, $\overline{a} = \frac{1}{n}\sum_i a_i$.

Metric	Formula
Mean Squared Error (MSE)	$\frac{(\widehat{m}_1 - a_1)^2 + \ldots + (\widehat{m}_n - a_n)^2}{n}$
Root Mean Squared Error ($RMSE$)	$\sqrt{\frac{(\widehat{m}_1 - a_1)^2 + \ldots + (\widehat{m}_n - a_n)^2}{n}}$
Mean Absolute Error (MAE)	$\frac{\lvert\widehat{m}_1 - a_1\rvert + \ldots + \lvert\widehat{m}_n - a_n\rvert}{n}$
Relative Squared Error (RSE)	$\frac{(\widehat{m}_1 - a_1)^2 + \ldots + (\widehat{m}_n - a_n)^2}{(a_1 - \overline{a})^2 + \ldots + (a_n - \overline{a})^2}$
Root Relative Squared Error ($RRSE$)	$\sqrt{\frac{(\widehat{m}_1 - a_1)^2 + \ldots + (\widehat{m}_n - a_n)^2}{(a_1 - \overline{a})^2 + \ldots + (a_n - \overline{a})^2}}$
Relative Absolute Error (RAE)	$\frac{\lvert\widehat{m}_1 - a_1\rvert + \ldots + \lvert\widehat{m}_n - a_n\rvert}{\lvert a_1 - \overline{a}\rvert + \ldots + \lvert a_n - \overline{a}\rvert}$

measured in the same dimension as the QoS value being predicted [24]. Mean Absolute Error (MAE) takes the average of the absolute values of the errors $\widehat{m}_i - a_i$ (i.e., $|\widehat{m}_i - a_i|$). In contrast to MSE, MAE is more robust against outliers.

To ease comparison, relative errors are computed. To this end, errors computed following one of the above metrics are normalized by the error of a very simple predictor (such as a simple average of past values). Relative error metrics include Relative Squared Error (RSE) and Relative Absolute Error (RAE). Again, the root squared relative error leads to a relative error that is of the same dimension as the predicted value.

Obviously, for the above metrics, smaller values indicate better prediction accuracy.

3.2 Limitations of Prediction Error Metrics

The prediction error, as expressed by metrics such as MAE and $RMSE$, has been used for SLA violation prediction [14,33]. Yet, using the prediction error has the limitation that it does not reveal the prediction accuracy in terms of whether failures (such as SLA violations or deviations from expected QoS values) are accurately predicted. However, such a prediction of failures is required in order to determine whether to adapt or not (see Section 2.3).

To illustrate the limitations of using the prediction error, let us discuss two "critical" cases: (1) Although the difference between the predicted and the actual

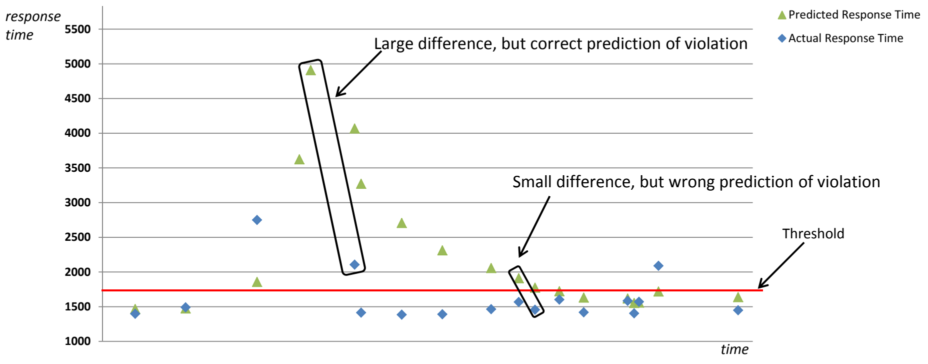


Fig. 3. Limitations of prediction error metrics (using response time prediction as example)

value may be relatively small, it may still imply an incorrect prediction of a failure; e.g., if the actual value is just below the QoS threshold and the predicted value is just above. (2) The difference between predicted and actual values may be relatively large, but may still lead to a correct prediction of a failure; e.g., if both the predicted value and actual value are above or below the threshold.

The diagram in Figure 3 shows these two "critical" cases for results from our experiments as described in Section 2.4. The large difference (on the left hand side of the diagram) is equal to 2802 ms but it still implies a correct prediction of a failure. The small difference (on the right hand side) is equal to 454 ms (i.e., only 16% of the large difference) but leads to an incorrect prediction of a failure.

This shortcoming can be addressed by accuracy metrics that consider failures and not numeric QoS values. They are introduced in the following section.

3.3 Assessing Binary Predictions

To know whether or not to trigger adaptations, not only the predicted QoS value is of interest but also whether or not it actually means a failure. Thus, in order to assess how accurately we can predict the need for adaptation, we need to take into account how accurately those failures can be predicted.

To this end, this section discusses metrics that can be used for assessing the accuracy of such "binary" predictions, i.e., predictions of failures and/or non-failures. These metrics are derived from the so called *contingency table*, which characterizes the four cases that can result from a binary prediction of failures (see Table 2).

Several metrics derived from the contingency table have been proposed in the literature. Table 3 provides a selection of these to highlight some of the more commonly used.

Precision (p) is defined as the ratio of correctly predicted failures to all predicted failures. Recall (r) is the ratio of correctly predicted failures to all actual failures. For the adaptation in service-oriented systems (see Section 2.3),

Table 2. Contingency Table

		Prediction	
		Failure	Non-failure
Actual	Failure	True Positive (TP)	False Negative (FN)
	Non-Failure	False Positive (FP)	True Negative (TN)

Table 3. Contingency table metrics based on [49]

Metric	Formula	Meaning
Precision (p)	$\frac{TP}{(TP+FP)}$	How many predicted failures were actual failures?
Recall(r)	$\frac{TP}{(TP+FN)}$	How many actual failures were correctly predicted as failures?
Specificity (s)	$\frac{TN}{(TN+FP)}$	How many actual non-failures were correctly predicted as non-failures?
False Positive Rate (fpr)	$\frac{FP}{(FP+TN)}$	How many predicted failures were actual non-failures?
Neg. Pred. Value (npv)	$\frac{TN}{(TN+TP)}$	How many predicted non-failures were actual non-failures?
Accuracy (a)	$\frac{(TP+TN)}{(TP+TN+FP+FN)}$	How many predictions were correct?
F-measure (F_β)	$\frac{(1+\beta^2)\cdot p\cdot r}{\beta^2\cdot p+r}$	Harmonic mean of p and r.

precision p can be used to assess incorrectly predicted adaptation needs, i.e., unnecessary adaptations. Higher precision implies less "false" alarms and thus less unnecessary adaptations. Similarly, recall r can be related to missed adaptations. Higher recall implies more actual failures being predicted and thus less missed adaptations.

In general, to perform well, a prediction technique must achieve both high precision and high recall. However, a trade-off exists between precision and recall. Improving precision, i.e., reducing the number of false positives, at the same time may result in worse recall, i.e., increase the number of false negatives [37,49]. Additionally, if a model always predicts failures, its recall will be 1 but the precision will be low. In this case, we cannot say that the model performs well. On the other hand, if a model predicts only one failure and the prediction turns out to be correct, the model's precision will be 1. Furthermore, a predictor that predicts non-failure for every instance will have precision 1, but recall evaluates to 0. Thus, here we cannot consider this model to have good accuracy either.

To reflect the trade-off between precision and recall in one single metric, the F-measure (F) has been proposed and used in the literature [37,49,36,59,27]. The F-measure is defined as the weighted harmonic mean of precision and recall, where precision is weighted by $\beta \geq 0$.

The F-measure is evenly balanced when $\beta = 1$. It favors precision when $\beta < 1$, and recall when $\beta > 1$. The higher the value of the F-measure, the better the accuracy of the prediction. Compared to the arithmetic mean, both precision and recall need to be high in order for the harmonic mean to be high. If precision and recall both equal zero, the F-measure is not defined, but the discontinuity can be removed and the F-measure to be defined as 0 in this case [49].

The shortcoming of the F-measure is that it does not take the true negative rate (tnr) into account, and metrics that consider this (see below) may be preferable to assess the accuracy of a binary prediction.

The false positive rate (fpr) is defined as the ratio of incorrectly predicted failures to the number of all non-failures. The smaller the false positive rate, the better, provided that the other metrics are not changed for the worse. False positive rate and true positive rate (i.e., recall) are often used in combination.

The negative predictive value (npv) is defined as the ratio of correctly predicted non-failures to all predicted non-failures. The higher the negative predictive value, the better the accuracy of the prediction. Following the same reasoning as for recall r, the negative predictive value npv can be related to missed adaptations. A higher negative predictive value implies more actual failures being predicted and thus less missed adaptations.

Specificity (s) is defined as the ratio of correctly predicted non-failures to all actual non-failures. Increasing specificity often worsens the negative predictive value, and thus, one can – again – use the harmonic mean to reflect the trade-off between them. Similar to precision p, specificity can be used to assess incorrectly predicted adaptation needs, i.e., unnecessary adaptations. Higher specificity implies less "false" alarms and thus less unnecessary adaptations.

Accuracy (a) is the ratio of correct predictions in comparison to all predictions performed[3]. However, Salfner et al. recommended not to use accuracy as the sole indicator of how well a prediction technique performs, as "due to the fact that failures usually are rare events [...] a strategy that always classifies the system to be nonfaulty can achieve excellent accuracy since it is right in most of the cases, although it does not catch any failure (recall is zero)" [49].

To conclude, in general, metrics that cover all of the four cases of the contingency table should be considered in order to gain a comprehensive picture of prediction accuracy. As a consequence we use the F and s metrics when we present our experimental results in the remainder of the chapter, as these cover all those four cases.

As a side note, when building a service-oriented system, service integrators will strive to choose third-party services with relatively low failure rates (informally,

[3] As a note on terminology, despite the concise definition of accuracy as a, we will use "accuracy" as a generic term (as already used in the beginning) in the remainder of the chapter, as we will not employ a during our further discussions.

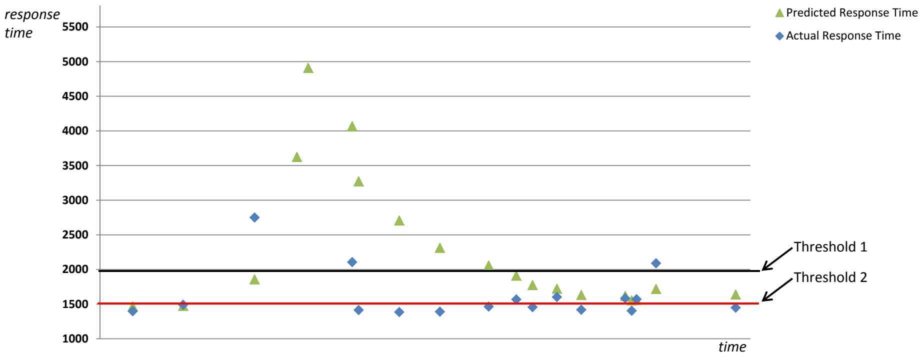


Fig. 4. Sensitivity of the contingency table to the threshold value for determining a failure

they will seek to employ reliable service providers). Thus, in service-oriented systems, the majority of observations will indicate non-failures, and, as a result, the negative predictive value, specificity and accuracy will always be high (as long as the prediction technique is not overly "anxious" to predict positives). For this reason, precision p and recall r (or the F-measure of p and r) may still be sufficient to evaluate predictors on these data sets.

3.4 Limitations of Contingency Table Metrics

Although binary metrics allow us to assess the accuracy of the need for proactive adaptations, they are sensitive to the threshold values used by the prediction models for determining failures during prediction.

Figure 4 shows the response times of a real service and its predicted QoS values using a time series prediction model (see Sections 2.2.2 and 4.1). Furthermore, it shows different values of thresholds. We can see that a slight increase in threshold value means that many false positives now become true negatives and vice versa. Using the data behind this figure, we can compute that Threshold 1 leads to $p = 0.17$ and $r = 0.33$, whereas Threshold 2 leads to $p = 0.43$ and $r = 0.86$. Although the difference in threshold values was just 500 ms, the differences in precision and recall values for the two thresholds are large.

To better assess the accuracy of a prediction model while reducing the impact of the threshold value used, we suggest (following the research literature [49]) assessing the accuracy using different threshold values. In addition, prediction error metrics should be employed and used along with the binary metrics to better understand how reliable the accuracy is reflected in the measurements. This should lead to more informed decisions such as to avoid missed or unnecessary adaptation triggers.

Figure 5 presents empirical data that show the accuracy of response time predictions[4] for the two services $s1$ and $s2$. The box plots on the left-hand side

[4] We used the prediction model Last (see Section 4.1).

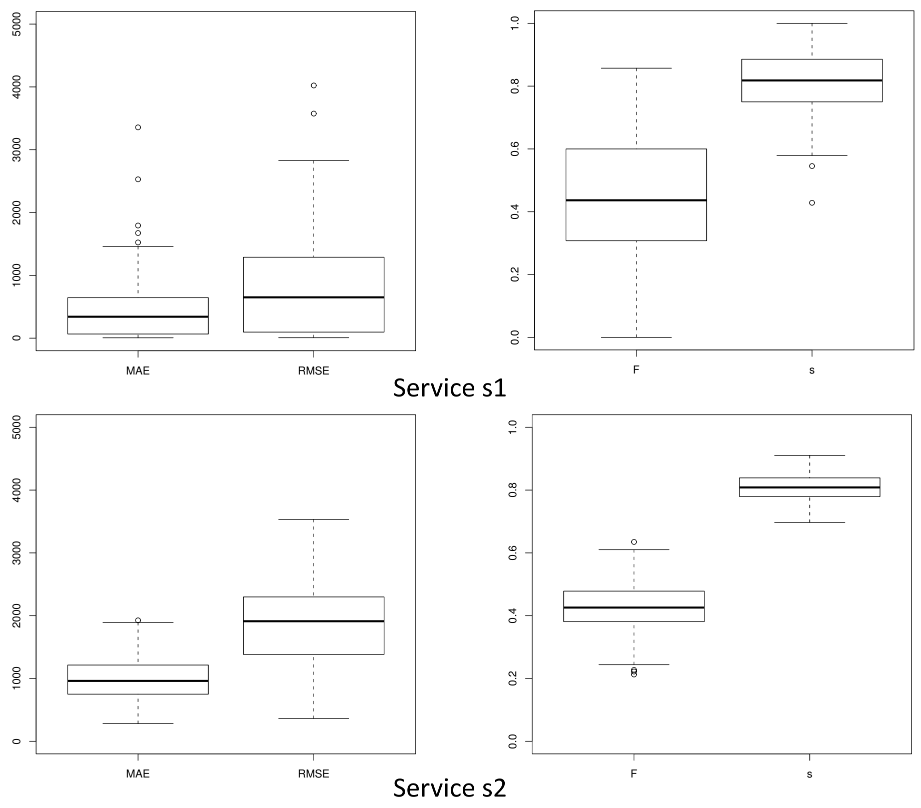


Fig. 5. Comparison of prediction error metrics with metrics that use threshold for determining accuracy

show the measured accuracy in terms of MAE and $RMSE$ (prediction error metrics from Section 3.2). Those are contrasted with the binary metrics F and s presented in Section 3.3.

The figure shows that although the prediction error for $s1$ (in terms of MAE and $RMSE$) is smaller than that of $s2$, accuracy in terms of F and s is almost equal for both $s1$ and $s2$. This means that these prediction error metrics did not reliably reflect the accuracy of predicting failures.

4 Ensuring Accurate Predictions

As introduced in Section 2.2.2, one important class of online quality prediction techniques for services are time series prediction models. In this section, we first review those prediction models in the light of ensuring accurate service quality predictions (Section 4.1). We identify their shortcomings when applied to service-oriented systems (Section 4.2) and propose and critically reflect on online testing as an alternative approach to achieve accuracy (Sections 4.3 and 4.4).

4.1 Time Series Prediction

Time series prediction models work on a series of past observations:

$$m_1, m_2, ..., m_i, ..., m_{n-1}, m_n,$$

with m_i being the observed QoS value at time point i. They predict the QoS value for time point $n+1$, i.e., $\widehat{m}_{n+1}$.

Some typical examples for such prediction models include:

– **Last**: This model uses the last observed value as the prediction value:

$$\widehat{m}_{n+1} = m_n$$

– *Windowed Mean*, $\mathbf{BM}(k)$: In this model, the arithmetic mean of the past k values $(1 \leq k \leq n)$ is used as the prediction value, where k is chosen to minimize prediction error:

$$\widehat{m}_{n+1} = \frac{1}{k} \sum_{i=0}^{k-1} m_{n-i}$$

– *Simple Exponential Smoothing*, $\mathbf{SEM}(\alpha)$: BM treats past observations equally. Conversely, SEM places more weight on more recent observations ($\alpha \in [0,1]$):

$$\widehat{m}_{n+1} = \alpha \cdot m_n + (1-\alpha) \cdot \widehat{m}_n$$

4.2 Limitations of Time Series Prediction

Time series predictors in general imply certain limitations when used for short-term prediction of service quality. As observed by Casolari et al. [12], there is a conflict between being responsive enough to changes in past observations and being too responsive and thus being perturbed by noise and fluctuations. On the one hand, if the smoothing of the past observations is too strong, this may lead to a delay in the predictions that is too excessive. Thus, a prediction model such as Last (only taking into account one previous observation) may be more responsive to abrupt changes in QoS. On the other hand, too little smoothing may lead to limited precision in highly variable scenarios, thus favoring models, such as BM or SEM.

Figure 6 shows a comparison of using the prediction models Last, BM, and SEM to predict QoS of third-party services for two differing usage settings. The experimental data has been collected using the experimental setup as described in Section 2.4.

Comparing the four prediction models used, there is no clear candidate which would outperform the others in terms of both the F and s metrics (see Section 3.3). In the first setting, all prediction models perform almost identical in terms of F (the boxplots significantly overlap), but Last outperforms the others in terms of s. In the second setting, SEM and BM(5) outperform Last in terms of F, but are

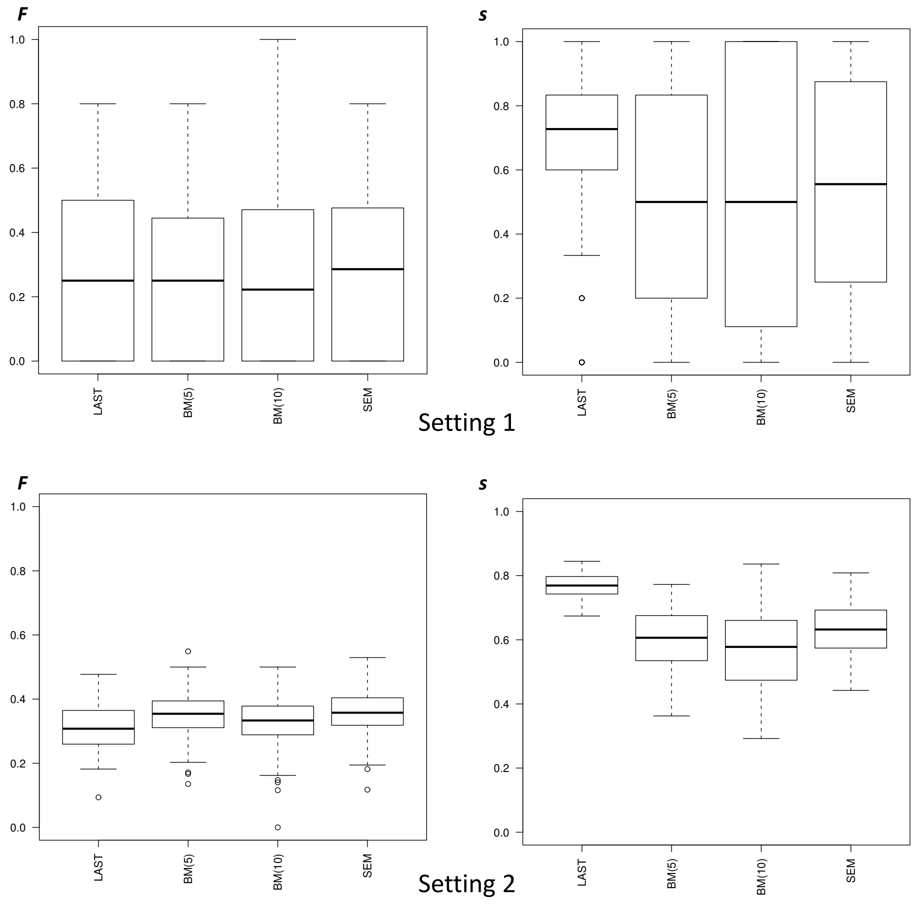


Fig. 6. Comparison of Time Series Prediction Models (using $\alpha = 0.3$ for SEM)

weaker than Last in terms of s. This may be an indicator of the aforementioned conflicts between prediction models, thus deserving further investigation.

When analyzing the accuracy of the prediction models in more detail, we can make a further observation. Figure 7 shows the prediction accuracy when clustering the results for the Last prediction model with respect to how frequently a service has been used (and thus monitored)[5].

As the data in Figure 7 indicates, the accuracy increases as the services are more frequently used and thus more frequent monitoring data (past observation) becomes available. This actually points to an important difference to more traditional computer-based systems. In contrast to those systems, monitoring data (and thus observations) in a service-oriented system might not arrive

[5] The other usage models show the very same behavior and we therefore do not repeat their diagrams here.

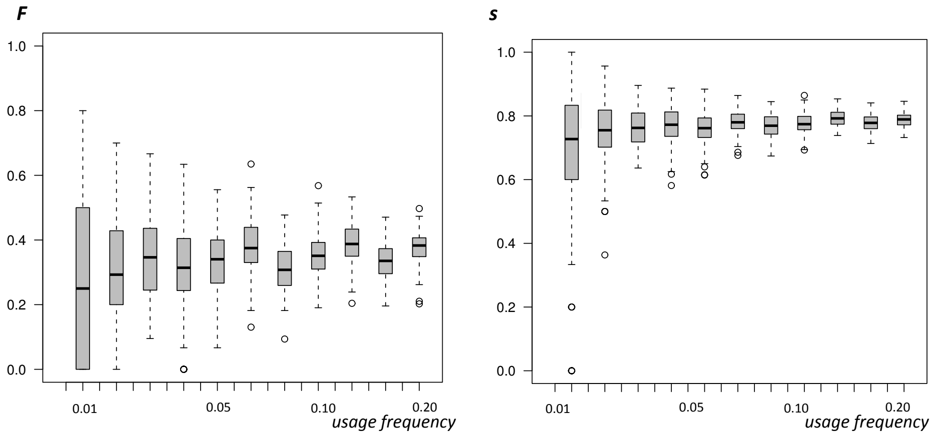


Fig. 7. Accuracy for Last prediction model depending on service usage frequency

periodically. This is due to the fact that only when a third-party service is actually being used, the monitoring mechanisms will observe how this service behaves (see Section 2.3) and whether there was a deviation in QoS. As a consequence, it may well happen that some predictions base their results on observations which are way outdated. Figure 3 in Section 3.2 shows a sample of how those observations and predictions are distributed along the time dimension.

To understand this further, we analyze the properties of the raw data set (from Cavallo et al. [14]) we used in our experiments. For a raw data set, predictions within a time window j (i.e., j steps ahead) in general can be performed with adequate accuracy if the auto-correlation function ($ACF(j)$) applied to the data set results in $|ACF(j)| \geq 0.3$ (see [10]). Figure 8 shows the the results of an auto-correlation analysis for two of the services from the Cavallo et al. data set.

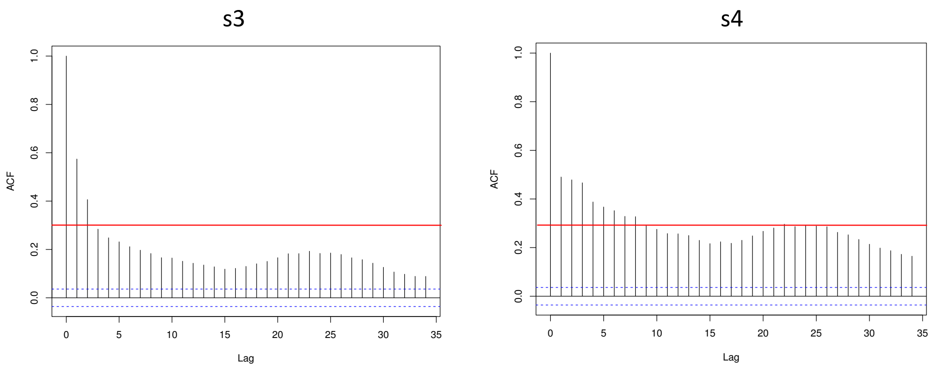


Fig. 8. Auto-correlation functions for two services used in experiments

As can be seen from these diagrams, the prediction window for which to achieve accurate predictions may diminish relatively quickly. In the case of service $s3$, this is already the case for $j = 3$. This means that if monitoring observations occur only very infrequently, the predictors in fact need to predict for long lags. Thus, the accuracy of any QoS prediction model may be severely limited in the setting of service-oriented systems.

Based on the above observations, we discuss an approach that aims to collect additional (i.e., more frequent) data points in order to improve prediction accuracy. Specifically, we will show that online testing for service-oriented systems allows systematically collecting such additional observations.

4.3 Online Testing

Most online quality prediction techniques for third-party services that have been presented in the literature (see Section 2.2.2) rely on monitoring to observe QoS values. However, monitoring is only able to retrieve QoS data when a service is actually being used during run-time. This means that monitoring is *passive* [47]. As we have seen in the previous section, the amount and timeliness of data may be limited as a result, restricting the accuracy of predictions.

Online testing, i.e., *actively* invoking services in parallel to their actual use, has been proposed as a quality assurance technique to complement passive monitoring [9,22,11].

In our previous work [50,45,40] we proposed augmenting monitoring data with online testing results. We provided results of initial, exploratory experiments that indicated that, in fact, accuracy of online quality predictions may be improved.

Figure 9 below shows the results of more comprehensive experiments (the setup of which has been summarized in Section 2.4). The white boxplots show the accuracy when augmenting monitoring with online testing data, the grey

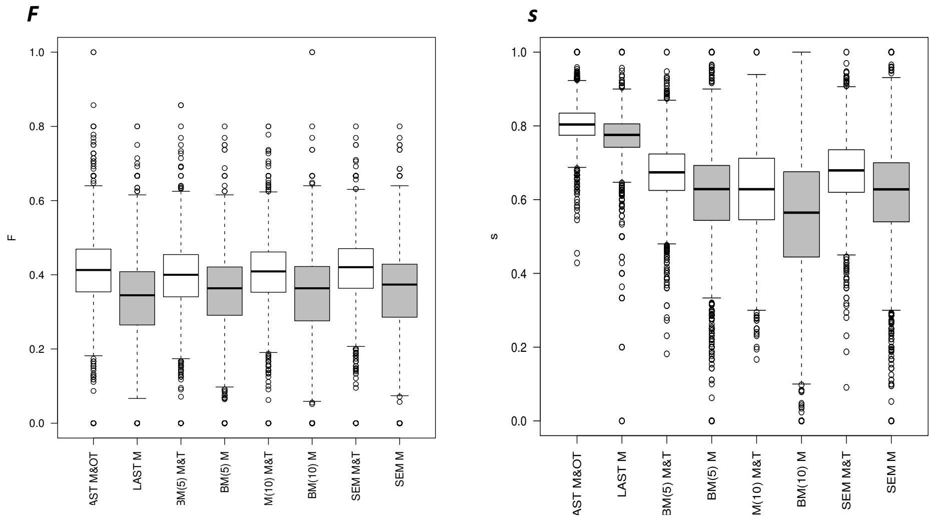


Fig. 9. Accuracy gains through online testing (M&T = monitoring data complemented by data from online testing; M = monitoring data only)

boxplots show the results when only using monitoring data. As can be seen, there is a clear improvement in prediction accuracy when employing additional data from online testing across all time series prediction models we used.

4.4 Limitations and Challenges of Online Testing

Although the accuracy gains that may be achieved by online testing are promising, the approach faces some limitations and challenges that are worth discussing.

Different factors may impact on the accuracy gains achieved by online testing. Figure 10 uses a sample of our experimental results to illustrate one such impact. The diagram compares the prediction accuracy using only monitoring data with the prediction accuracy when augmenting monitoring data with data from online testing. As can be observed, gains in prediction accuracy decrease as more frequent monitoring data is available, i.e., the more frequently the service is actually being used.

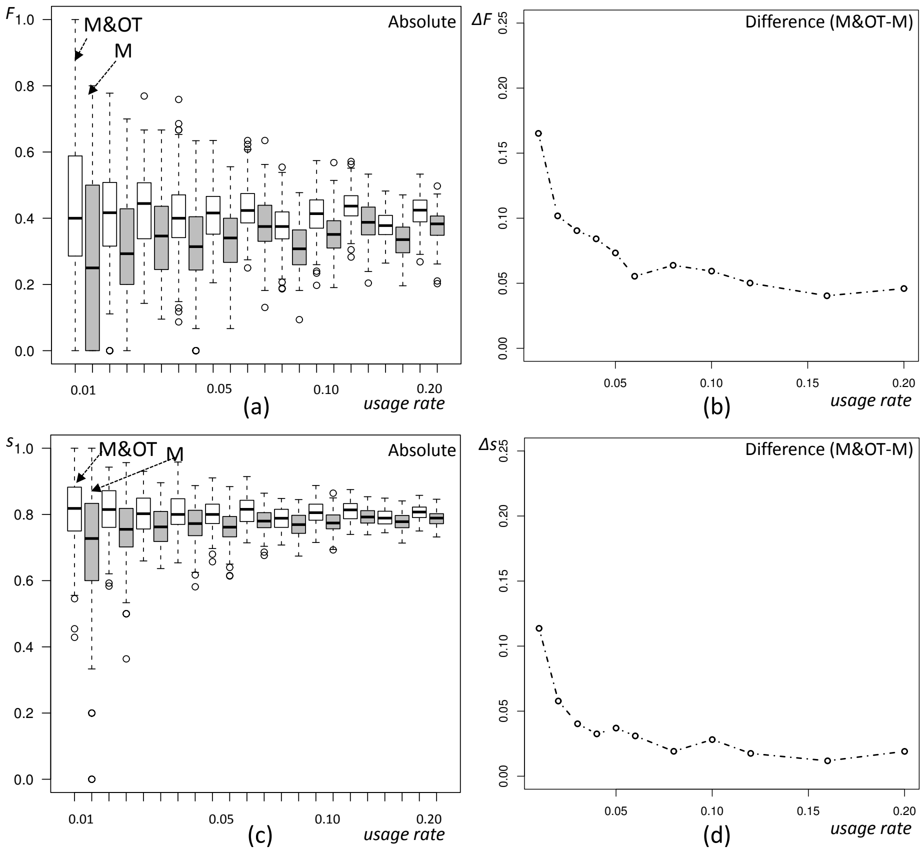


Fig. 10. Accuracy gains for different usage rates (M&T = monitoring data complemented by data from online testing; M = monitoring data only)

Moreover, online testing may imply additional costs for service integrators. For example, when testing a pay-per-use service, each invocation of the service is charged [16]. This means that there may be situations in which the benefits of achieving higher prediction accuracy may not pay off with respect to the additional costs imposed by online testing. As an example, if it takes 1,000 tests to avoid one more QoS violation (when compared to monitoring), online testing most probably will not pay off.

In addition to the limitations mentioned above, online testing faces some well-known technical challenges. For instance, online testing can be problematic for services that – when tested – produce side effects such as shipping items or charging credit cards. Additionally, online testing could be problematic for services with limited resources. Online testing means additional load, which may interfere with normal operations, thereby degrading the QoS performance of the service provider. There are several proposals to address these issues [11,21,17]. In general, it is proposed to offer test interfaces or sandboxes allowing a service to be executed in a special testing environment or configuration mode. Yet, although this allows the functionality of a service to be fully exercised in isolation from the real production environment and databases, it remains questionable how representative the testing results will be for the actual service in operation.

5 Conclusions and Perspectives

Proactive adaptation of service-oriented systems relies on the ability to perform short-term predictions of service quality. Only if these predictions are accurate, proactive adaptation will be of use in practice. However, as we have seen in this chapter, achieving accurate online quality predictions is a challenging endeavor.

In this chapter we have barely scratched the surface of what remains ahead of us in this field of research. Looking ahead, current trends such as the "Future Internet" [56,1] and "Cloud Computing" [5] are worth noting. On the one hand, the "Future Internet", through the convergence between the Internet of Services (IoS) and the Internet of Things (IoT), will foster and ease cross-organizational data exchange and integration of IT systems with real-world processes. As an example, the information associated with IoT-sensors will be accessible using service technology, such that their functionalities can be discovered and accessed over the Internet (in fact, proposals in this direction have already been made [54,23,53]). On the other hand, "Cloud Computing" will make computing and networking resources accessible in a much more flexible and comprehensive way. Ultimately, these trends will lead to even higher levels of dynamicity than those we face today in service-oriented systems. Future software systems thus will require even stronger adaptation and prediction capabilities in order to become resilient to changes in such highly dynamic environments. The interested reader can find a more in-depth discussion of those challenges and related research issues in [7,42].

Acknowledgments. Many of the findings reported in this chapter have been stimulated by discussions with researchers from S-Cube, the European Network of Excellence on "Software Services and Systems", as well as FInest, the EU Integrated Project on "Future Internet Enabled Transport & Logistics". To all of these colleagues we extend our deepest gratitude.

The first author would like to thank the DEEP-SE group at Politecnio di Milano, as well as the organizers of the ASAS workshop at ESEC/FSE 2012 for providing a platform to discuss the initial ideas (published as [40]), which have been extended and refined in this book chapter.

The research leading to these results has received funding from the European Unions's Seventh Framework Programme FP7/2007-2013 under grant agreements 215483 (S-Cube), 285598 (FInest) and 285248 (FI-WARE).

References

1. Álvarez, F., Cleary, F., Daras, P., Domingue, J., Galis, A., Garcia, A., Gavras, A., Karnourskos, S., Krco, S., Li, M.-S., Lotz, V., Müller, H., Salvadori, E., Sassen, A.-M., Schaffers, H., Stiller, B., Tselentis, G., Turkama, P., Zahariadis, T. (eds.): Future Internet Assembly 2012: From Promises to Reality. LNCS, vol. 7281. Springer, Heidelberg (2012)
2. Amin, A., Colman, A., Grunske, L.: An approach to forecasting QoS attributes of web services based on ARIMA and GARCH models. In: Proceedings of the IEEE International Conference on Web Services (ICWS 2012), pp. 74–81. IEEE (2012)
3. Amin, A., Grunske, L., Colman, A.: An automated approach to forecasting QoS attributes based on linear and non-linear time series modeling. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012. IEEE/ACM (to appear, 2012)
4. Apt, K.: Principles of Constraint Programming. Cambridge University Press (2003)
5. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Commun. ACM 53, 50–58 (2010)
6. Aschoff, R., Zisman, A.: QoS-Driven Proactive Adaptation of Service Composition. In: Kappel, G., Maamar, Z., Motahari-Nezhad, H.R. (eds.) ICSOC 2011. LNCS, vol. 7084, pp. 421–435. Springer, Heidelberg (2011)
7. Baresi, L., Georgantas, N., Hamann, K., Issarny, V., Lamersdorf, W., Metzger, A., Pernici, B.: Emerging research themes in services-oriented systems. In: Proceedings of the SRII 2012 Global Conference. Conference Publishing Service (CPS), IEEE Computer Society (2012)
8. Bertolino, A., Angelis, G.D., Polini, A.: (role)CAST: A framework for on-line service testing. In: Proceedings of the 7th International Conference on Web Information Systems and Technologies, WEBIST 2011, pp. 13–18. SciTePress (2011)
9. Bertolino, A., De Angelis, G., Kellomaki, S., Polini, A.: Enhancing service federation trustworthiness through online testing. Computer 45(1), 66–72 (2012)
10. Brockwell, P., Davis, R.: Time series: theory and methods. Springer (2009)
11. Canfora, G., Di Penta, M.: Testing services and service-centric systems: Challenges and opportunities. IT Professional 8, 10–17 (2006)
12. Casolari, S., Colajanni, M.: Short-term prediction models for server management in internet-based contexts. Decision Support Systems 48(1), 212–223 (2009)

13. Cassales Marquezan, C., Metzger, A., Pohl, K., Engen, V., Boniface, M., Phillips, S.C., Zlatev, Z.: Adaptive future internet applications: Opportunities and challenges for adaptive web services technology. In: Ortiz, G., Cubo, J. (eds.) Adaptive Web Services for Modular and Reusable Software Development: Tactics and Solution, pp. 333–353. IGI Global (2013)
14. Cavallo, B., Di Penta, M., Canfora, G.: An empirical comparison of methods to support QoS-aware service selection. In: Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems, PESOS 2010, pp. 64–70. ACM (2010)
15. Cheng, B., et al.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
16. Di Penta, M., Bruno, M., Esposito, G., et al.: Web Services Regression Testing. In: Baresi, L., Di Nitto, E. (eds.) Test and Analysis of Web Services, pp. 205–234. Springer (2007)
17. Dranidis, D., Metzger, A., Kourtesis, D.: Enabling Proactive Adaptation through Just-in-Time Testing of Conversational Services. In: Di Nitto, E., Yahyapour, R. (eds.) ServiceWave 2010. LNCS, vol. 6481, pp. 63–75. Springer, Heidelberg (2010)
18. Ejarque, J., Micsik, A., Sirvent, R., Pallinger, P., Kovacs, L., Badia, R.: Semantic resource allocation with historical data based predictions. In: Proceedings of the 1st International Conference on Cloud Computing, GRIDs, and Virtualization, Cloud Computing 2010 (2010)
19. Engel, Y., Etzion, O.: Towards proactive event-driven computing. In: Proceedings of the 5th ACM International Conference on Distributed Event-Based System, DEBS 2011, pp. 125–136. ACM (2011)
20. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 341–350. ACM (2011)
21. González, A., Piel, E., Gross, H.G.: A model for the measurement of the runtime testability of component-based systems. In: Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2009, pp. 19–28. IEEE Computer Society (2009)
22. Greiler, M., Gross, H.G., van Deursen, A.: Evaluation of online testing for services: a case study. In: Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems, PESOS 2010, pp. 36–42. ACM (2010)
23. Guinard, D., Trifa, V., Karnouskos, S., Spiess, P., Savio, D.: Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. IEEE Transactions on Services Computing 3, 223–235 (2010)
24. Han, J., Kamber, M.: Data Mining: Concepts and Techniques, 2nd edn. Morgan Kaufmann (2005)
25. Haykin, S.: Neural Networks and Learning Machines: A Comprehensive Foundation, 3rd edn. Prentice-Hall (2008)
26. Hielscher, J., Kazhamiakin, R., Metzger, A., Pistore, M.: A Framework for Proactive Self-adaptation of Service-Based Applications Based on Online Testing. In: Mähönen, P., Pohl, K., Priol, T. (eds.) ServiceWave 2008. LNCS, vol. 5377, pp. 122–133. Springer, Heidelberg (2008)
27. Huang, L., Ke, X., Wong, K., Mankovskii, S.: Symptom-based problem determination using log data abstraction. In: Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research, CASCON 2010, pp. 313–326. ACM (2010)

28. Ivanovic, D., Carro, M., Hermenegildo, M.: Towards data-aware qos-driven adaptation for service orchestrations. In: Proceedings of the IEEE International Conference on Web Services, ICWS 2010, pp. 107–114. IEEE Computer Society (2010)
29. Ivanović, D., Carro, M., Hermenegildo, M.: Constraint-Based Runtime Prediction of SLA Violations in Service Orchestrations. In: Kappel, G., Maamar, Z., Motahari-Nezhad, H.R. (eds.) ICSOC 2011. LNCS, vol. 7084, pp. 62–76. Springer, Heidelberg (2011)
30. Ivanović, D., Treiber, M., Carro, M., Dustdar, S.: Building Dynamic Models of Service Compositions with Simulation of Provision Resources. In: Parsons, J., Saeki, M., Shoval, P., Woo, C., Wand, Y. (eds.) ER 2010. LNCS, vol. 6412, pp. 288–301. Springer, Heidelberg (2010)
31. Jamoussi, Y., Driss, M., Jézéquel, J.M., Ben Ghézala, H.: QoS assurance for service-based applications using discrete-event simulation. IJCSI International Journal of Computer Science Issues 7(4) (2010)
32. Josuttis, N.: SOA in Practice: The Art of Distributed System Design. O'Reilly Media (2007)
33. Leitner, P., Michlmayr, A., Rosenberg, F., Dustdar, S.: Monitoring, prediction and prevention of SLA violations in composite services. In: Proceedings of the IEEE International Conference on Web Services, ICWS 2010, pp. 369–376. IEEE Computer Society (2010)
34. Leitner, P., Wetzstein, B., Karastoyanova, D., Hummer, W., Dustdar, S., Leymann, F.: Preventing SLA Violations in Service Compositions Using Aspect-Based Fragment Substitution. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 365–380. Springer, Heidelberg (2010)
35. de Lemos, R., et al.: Software Engineering for Self-Adpaptive Systems: A second Research Roadmap. In: de Lemos, R., Giese, H., Müller, H., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems. Dagstuhl Seminar Proceedings, vol. 10431, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2011)
36. Liang, Y., Zhang, Y., Xiong, H., Sahoo, R.: Failure prediction in IBM BlueGene/L event logs. In: Proceedings of the 2007 Seventh IEEE International Conference on Data Mining, ICDM 2007, pp. 583–588. IEEE Computer Society (2007)
37. Ma, Y., Cukic, B.: Adequate and precise evaluation of quality models in software engineering studies. In: Proceedings of the Third International Workshop on Predictor Models in Software Engineering, PROMISE 2007. IEEE Computer Society (2007)
38. Mancioppi, M.: Consolidated and updated state of the art report on Service-Based Applications (CD-IA-1.1.7). Tech. rep., S-Cube Network of Excellence (November 2011)
39. Menasce, D.A., Almeida, V.: Capacity Planning for Web Services: metrics, models, and methods, 1st edn. Prentice Hall PTR, Upper Saddle River (2001)
40. Metzger, A.: Towards accurate failure prediction for the proactive adaptation of service-oriented systems. In: Proceedings of the 8th Workshop on Assurances for Self-Adaptive Systems, ASAS 2011, pp. 18–23. ACM (2011) (invited); collocated with ESEC 2011
41. Metzger, A., Cassales Marquezan, C.: Future Internet Apps: The Next Wave of Adaptive Service-Oriented Systems? In: Abramowicz, W., Llorente, I.M., Surridge, M., Zisman, A., Vayssière, J. (eds.) ServiceWave 2011. LNCS, vol. 6994, pp. 230–241. Springer, Heidelberg (2011)

42. Metzger, A., Chi, C.H., Engel, Y., Marconi, A.: Research challenges on online service quality prediction for proactive adaptation. In: Proceedings of the ICSE 2012 Workshop on European Software Services and Systems Research – Results and Challenges, S-Cube (2012)
43. Metzger, A., Di Nitto, E.: Addressing highly dynamic changes in service-oriented systems: Towards agile evolution and adaptation. In: Wang, X., Ali, N., Ramos, I., Vidgen, R. (eds.) Agile and Lean Service-Oriented Development: Foundations, Theory and Practice. IGI Global (2012)
44. Metzger, A., Franklin, R., Engel, Y.: Predictive monitoring of heterogeneous service-oriented business networks: The transport and logistics case. In: SRII 2012 Global Conference. Conference Publishing Service (CPS). IEEE Computer Society (2012)
45. Metzger, A., Sammodi, O., Pohl, K., Rzepka, M.: Towards pro-active adaptation with confidence: Augmenting service monitoring with online testing. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010), pp. 20–28. ACM (2010)
46. Nitto, E.D., Ghezzi, C., Metzger, A., Papazoglou, M.P., Pohl, K.: A journey to highly dynamic, self-adaptive service-based applications. Automated Software Engineering 15(3-4), 313–341 (2008)
47. Papazoglou, M., Pohl, K., Parkin, M., Metzger, A. (eds.): Service Research Challenges and Solutions for the Future Internet: Towards Mechanisms and Methods for Engineering, Managing, and Adapting Service-Based Systems. Springer (2010)
48. Quinlan, J.R.: Induction of decision trees. Machine Learning 1, 81–106 (1986)
49. Salfner, F., Lenk, M., Malek, M.: A survey of online failure prediction methods. ACM Computing Surveys 42(3), 10:1–10:42 (2010)
50. Sammodi, O., Metzger, A., Franch, X., Oriol, M., Marco, J., Pohl, K.: Usage-based online testing for proactive adaptation of service-based applications (short). In: Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference (COMPSAC 2011), pp. 582–587. IEEE Computer Society (2011)
51. Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., Teneketzis, D.: Diagnosability of discrete-event systems. IEEE Transactions on Automatic Control 40(9), 1555–1575 (1995)
52. Schmieders, E., Metzger, A.: Preventing Performance Violations of Service Compositions Using Assumption-Based Run-Time Verification. In: Abramowicz, W., Llorente, I.M., Surridge, M., Zisman, A., Vayssière, J. (eds.) ServiceWave 2011. LNCS, vol. 6994, pp. 194–205. Springer, Heidelberg (2011)
53. Shelby, Z.: Embedded web services. IEEE Wireless Communications 17(6), 52–57 (2010)
54. Spiess, P., Karnouskos, S., Guinard, D., Savio, D., Baecker, O., de Souza, L.M.S., Trifa, V.: Soa-based integration of the internet of things in enterprise services. In: Proceedings of the IEEE International Conference on Web Services (ICWS 2009), pp. 968–975. IEEE Computer Society (2009)
55. Tsai, W.T., Zhou, X., Chen, Y., Bai, X.: On testing and evaluating service-oriented software. IEEE Computer 41(8), 40–46 (2008)
56. Tselentis, G., Domingue, J., Galis, A., Gavras, A., Hausheer, D.: Towards the Future Internet: A European Research Perspective. IOS Press, Amsterdam (2009)
57. Witten, I.H., Frank, E.: Data mining: practical machine learning tools and techniques, 2nd edn. Elsevier, Morgan Kaufman, Amsterdam (2005)

58. Yau, S.S., An, H.G.: Software engineering meets services and cloud computing. IEEE Computer 44(10), 47–53 (2011)
59. Yu, L., Zheng, Z., Lan, Z., Coghlan, S.: Practical online failure prediction for blue gene/p: Period-based vs event-driven. In: Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSNW 2011), pp. 259–264. IEEE Computer Society (2011)
60. Zeng, L., Lingenfelder, C., Lei, H., Chang, H.: Event-Driven Quality of Service Prediction. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 147–161. Springer, Heidelberg (2008)

Failure Avoidance in Configurable Systems through Feature Locality

Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer

University of Nebraska-Lincoln, Lincoln NE 68588-0115, USA
{bgarvin,myra,dwyer}@cse.unl.edu

Abstract. Despite the best efforts of software engineers, faults still escape into deployed software. Developers need time to prepare and distribute fixes, and in the interim, deployments must either avoid failures or endure their consequences. Self-adaptive systems—systems that adapt to changes internally, in requirements, and in a dynamic environment—can often handle these challenges automatically, depending on the nature of the failures.

Those self-adaptive systems where functional features can be added or removed also constitute configurable systems. Configurable software is known to suffer from failures that appear only under certain feature combinations, and these failures are particularly challenging for testers, who must find suitable configurations as well as inputs to detect them. However, these elusive failures seem well suited for avoidance by self-adaptation. We need only find an alternative configuration that precludes the failure without derailing the current use case.

This work investigates that possibility, along with some further conjectures: that the failures that are sensitive to a system's configuration depend on similar feature combinations—a phenomenon we call feature-locality—that this locality can be combined with historical data to predict failure-prone configurations and reconfiguration workarounds, and that these workarounds rarely lead the system out of one failure and into another. In a case study on 128 failures reported against released versions of an open source configurable system, and 16 failures discovered through a state-of-the-art testing tool, plus several thousand tests cases, we find evidence to support all of these hypotheses.

Keywords: self-adaptive software, highly configurable systems, failure avoidance, fault tolerance, reconfiguration, workarounds, software testing.

1 Introduction

Self-adaptive systems are growing in interest as an alternative architectural model when the goal is to ensure continuous operation under a variety of environments [1–6]. Adaptations force the system to reconfigure in order to increase reliability or performance and are triggered whenever environmental conditions necessitate change. The traditional adaptive feedback loop involves collecting and analyzing data, deciding what adaptation to use, and then acting on this decision by reconfiguring the system.

Recent research on *self-adaptation* and *fault tolerance* has developed techniques for preserving functionality in the face of deployed faults [3, 7–11]. Work in these two areas has produced a variety of approaches: exploiting redundancy in the architecture or implementation [1,6], directly modifying the source code [12,13], changing the architectural components or connectors [2,4,14], constructing adapters or wrappers to fix interoperability issues between components [3, 15], and transitioning to a set of precomputed "good" states when an error state is detected [7,16].

While reliability, a concern of self-adaptive systems and the focus of most of these techniques, reflects the collection of all possible failures, we might also focus on individual, discrete failures. A model for single failures is easier to build, because we need only replay failing test cases under various configurations. If different failures occur under similar configurations, then the combined model for a small number of failures will be a good approximation of the more general reliability model, and can be learned relatively quickly. The system can then consult the model to forbid troublesome configurations. This is a particular kind of *fault treatment* or *fault handling*, as it avoids error states rather than correcting them [17, 18], but, unlike most fault treatment schemes, it does not alter code. Therefore, throughout the rest of the paper we will use the term *failure avoidance* to reinforce the idea that the faults remain, and can still manifest as failures; it is only that we avoid paths that have been involved in prior failures.

To explore the viability of this idea, we examine a closely related area of research, validating *highly-configurable systems*—systems with features that can be added and removed [19]. Such systems may contain faults that cannot be exposed under every choice of features, and the prohibitive cost of testing all choices means that these faults stand a higher chance of eluding testers [20,21]. These types of systems, while not necessarily adaptive, have similar characteristics in that reconfigurations change the way executions occur while preserving a core set of functionality. For instance, FeatUre-oriented Self-adaptatION, (FUSION) [22], models adaptive systems in the same way we model highly-configurable systems, and it uses the notation of feature models [19] to represent adaptations. Moreover, in the FUSION case study, online learning is successful even though it identifies only a portion of features as relevant to the utility function. Correspondingly, there is some evidence in the software testing community that failures are dependent on only small combinations of features [5, 23, 24].

In this work, we build on the ideas of Hassan et al. [25] and Kim et al. [26], and hypothesize that failures have what we term *feature locality*, a tendency to depend on similar combinations of features. Under that hypothesis, configuration choices that avoid one failure are likely to avoid others, and maintaining a history of failures will allow us both to avoid them as we reconfigure and to select potential reconfiguration workarounds. In a preliminary study [27], we found feature locality across three releases of a well studied compiler collection, GCC. This paper expands our case study by using a state-of-the-art automated test generator [28], examining a new set of failures, and evaluating the robustness of reconfigurations, thereby reducing the threats to validity that arose from that study's limited test suite. We simulate online adaptations by reconstructing the failure timeline, and find that (1) very few features impact the visibility of any one failure, (2) within five reconfigurations a knowledge of failure history increases our effectiveness more than eight times over an uninformed strategy, (3) as failure history accumulates we need fewer reconfiguration attempts to avoid new failures, and (4) excepting one anomaly, reconfiguration workarounds hardly ever cause additional failures. These results suggest that feature locality exists and may be useful for self-adaptive software.

The contributions of this work are:

– A presentation of feature locality and its potential impact on ensuring dependability during reconfiguration.
– Algorithms for history-based reconfiguration to avoid and recover from failures.
– A case study to evaluate the existence of feature locality and our ability to exploit it on a set of failures detected in the field.

The rest of this paper is laid out as follows: In the next section we introduce background through a motivating example and discuss related work. We present our hypothesis in Section 3, Section 4 details our case study, and Section 5 concludes and highlights opportunities for future work.

2 Background and Related Work

Our hypotheses and technique draw from two areas of software engineering: the research on dynamically reconfigurable systems, autonomic systems, adaptation for correctness, and fault tolerance, and the work on prediction schemes for fault proneness. In this section, with the help of an example from NASA's Mars Exploration Program, we introduce the relevant terminology and illustrate the connections between this prior work and failure avoidance.

2.1 The Spirit Sol 18 Anomaly

Only 18 Mars solar days (sols, each approximately one Earth day) after landing, NASA's Mars rover, Spirit [29], encountered a nearly mission-ending software failure. The symptoms began with failed communications on the rover's two independent channels and eventually developed into intermittent, babbled transmissions along with an inability to obey basic commands. Not until sol 20 could NASA obtain crucial diagnostic information, including a health update packet, which showed signs of multiple reboots, a low battery, and a high internal temperature. Spirit was stuck in an endless reboot cycle and therefore failing to sleep; it risked running out of power or overheating.

The situation persisted through sol 21. Then the team at NASA managed a reboot with most of the flash file system disabled, and the rover started accepting commands consistently. Recovery became a possibility, though Spirit had to be put back in "crippled" mode every Martian morning.

In the meantime, NASA engineers strove to isolate the responsible software fault; although they were now able to avoid the failures, the root cause remained. On sol 71 the fault was found: a NASA-built component had expected the third-party file system to deallocate space as files were deleted, but deallocation only occurred when the enclosing directories were removed. As such, the file system had been bloated beyond a mountable size, and during boot, the failed mount from flash memory would trigger the default recovery action: a reboot. By sol 98 a fix was finished and installed.

In summary, the reconfiguration, which was found in three days through trial and error, meant that Spirit survived with limited functionality until the fix was delivered two and a half months later.

2.2 Terminology

Spirit was deployed as a single system, but its functionality was divided among *features*: the software and hardware components governing power, temperature, radios, sensors, motors, actuators, autonomous navigation, etc. Because most of these features could be enabled or disabled, Spirit constituted a *highly-configurable system*. However, like many highly-configurable systems, some features depended on or conflicted with others, so not every *configuration*—that is, not every choice of features—was valid. Under NASA's workaround for the anomaly, *feature constraints*, dependencies between features, disallowed most of Spirit's functionality because flash memory was disabled. (Even more functionality would have been lost without the workaround: Spirit would have been inoperable and eventually unrecoverable.)

A configurable system's features and feature constraints are usually represented compactly in a *feature model*. There are several languages for expressing a feature model in a form that mirrors the organization of functionality [30–32], but for our purposes we translate from these languages to a more uniform, relational form by following the process in [33]. The result is a sequence of configuration choices, each of which has a set of mutually exclusive options. For example, Spirit's solar panels and batteries, along with their associated software, can be toggled independently. Thus, the transformation creates one choice between active solar panels and a newly-introduced "null" feature, and it produces a similar choice for the batteries. Here we say that there are two *feature groups*, each containing two alternative features.

A sequence of independent feature choices makes for a structured model, but not one that is expressive enough to encode most feature constraints. Hence, as detailed in [34], we define one boolean variable for every feature and treat each configuration as an assignment to these variables; a variable is assigned true if and only if the configuration includes the corresponding feature. Then the restrictions on legal configurations can be expressed as a propositional formula. For instance, suppose we define P to mean that Spirit's solar panels are active and Q to mean that Spirit's batteries are providing energy. For Spirit to be in operation, $P \vee Q$ must hold, or else there will be no power supply.

Like other systems, when a highly-configurable system observably deviates from its requirements, we say that it has encountered a *failure* [35]. Spirit rebooting, nearly overheating, ignoring commands, and returning garbled data, for instance, constitutes a failure. We will reserve the term *fault* and its synonym *bug* for flaws in the system that make failures possible [35].

If the system would have met its requirements had it been configured differently, we say that the failure it encountered was *reconfiguration-avoidable*. Importantly, whether a failure is reconfiguration-avoidable depends on which requirements we are talking about; reconfiguration workarounds must sacrifice functionality to attain correctness. In NASA's race to save Spirit, for instance, the rover met its survival requirement once engineers eliminated the flash memory mount. But if NASA had needed the rover's more advanced scientific abilities to stay active, it is unlikely that the failure could have been avoided.

2.3 Adaptation for Correctness

For Spirit's development team, it was worthwhile to give special attention to a single deployment. That's not the case for most highly-configurable systems; we expect many deployments, perhaps in a variety of configurations. If we are to apply the lessons from the rover scenario elsewhere, recovery and failure avoidance must be at least partially automated and, if possible, leverage information from other deployments.

Systems that employ such automation—those that have the capacity to monitor their environment and behavior and then react—are called *autonomic* or *self-adaptive* [8, 11]. There is a large body of work describing such adaptation in both the hardware and the software domain. Researchers have considered systems designed to respond to poor performance [2], security vulnerabilities [13], architectural mismatches [2], misconfiguration [3,4], interoperability issues [3,15], and functional failures [4, 16, 36], all without human intervention. Additional studies have produced methods for validating these designs in safety-critical systems [37] and dynamic software product lines (SPLs) [5, 24].

Throughout all of this work the adaptation process can be divided into four activities: *monitoring*, *detecting*, *deciding*, and *acting* [8]. Monitoring is responsible for sensing the program's environment and tracking its behaviors. The data it gathers feeds into detection code, which determines whether the observations should force a change in the system's state. If a modification is warranted, the decision routines are invoked to choose a response, in this case a reconfiguration. The action phase carries out the decision code's choice.

We are primarily interested in the decision phase of systems that combat functional failures, *self-healing systems* [1–6]. Once self-healing software has determined that a failure has occurred, it must decide how to restore the system to a known good state, retry the failed task, and avoid the failure on future tasks. We concentrate on the last two choices.

Our approach can be viewed as an application of fault tolerance [9, 10] or more specifically, of fault treatment [17, 18, 38]. Fault tolerant systems detect error states caused by faults, and then return the system to an error-free state, or patch the system, or contain the error state, thereby avoiding the propagation of the fault as a failure. They often use replication (through techniques such as N-versioning) and online voting to aid recovery.

If we consider different system configurations that support the same functionality as different versions of a system, then our technique is also selecting a version that is correct. We do not, however, explicitly change the state of the system while running, and do not prevent error propagation dynamically. Instead, in this work, we analyze and reconfigure after a particular failure is seen. Our analysis associates the failures with particular sets of features, and then removes those feature combinations from the system. The result is the removal of potential paths to failures in subsequent executions of the system.

Fault treatment is concerned with avoiding future faults and is often achieved through code modification. For example, the work of de Lemos and Fiadeiro [38], like ours does not modify code, but instead suggests a fault treatment architecture with a flexible component connector structure to allow components to be easily replaced. They do not, however, provide algorithms to learn specific feature combinations that lead to failures, as we do.

Our approach extends the intuition pioneered by Hassan et al. [25] and refined by Kim et al. [26]: just as memory accesses exhibit spatial and temporal locality patterns that can be exploited by a cache, fault-introducing changes to a system's source code also demonstrate locality. The latter work points out four forms of locality exhibited by faults, two that refer to the time of changes and two that refer to their locations in the source code. We hypothesize a related locality, where a small set of feature combinations in a system's configuration space (rather than its change history or source code) are associated with failures (rather than faults), i.e. we see a locality of failures in the configuration space.

3 Technique for Avoidance

Our follow-on conjecture to this hypothesis is that we can learn from failures to guard against and recover from later ones. We view this process as a special case of fault tolerance, to be implemented in an auxiliary function on top of the decision phase in the normal self-adaptive loop. In a situation like Spirit's, with, at the time, only one active deployment, such learning would often come too late to be very useful. But other systems, like desktop applications, web servers, and embedded software, which have many concurrent deployments in similar circumstances, provide an opportunity to confine problems to the few deployments where they were first uncovered.

To evaluate the hypothesis, we make several simplifying assumptions. First, we assume that an explicit feature model is available to the software at runtime. We take as given the mechanisms to detect failures and to maintain state upon reconfiguration. Then, for the purposes of the study, we take avoidance of the discrete failure events to outweigh all other concerns measured by the utility function that guides adaptations. However, we do capture some of the tradeoff between functionality and correctness by dividing feature groups into those that can be modified and those that must not change. Finally, we assume that the input stream can be broken up into segments, which correspond to the *test cases* that we use in the remaining sections, and that segments provoke failures deterministically.

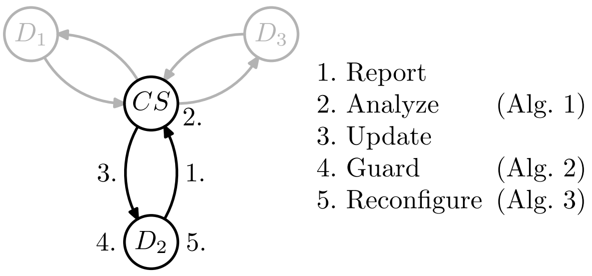


Fig. 1. Exploiting History

Figure 1 illustrates our approach at a high level. Because it is black-box, our technique must learn about the system by encountering failures in the field. But it would be wasteful if every deployment of the system had to see each failure, so instead we establish a central store to relay failure information between deployments. In the figure, the store is labeled CS, and each deployment is designated by a subscripted D. Reporting of failures in deployments may be automatic or there may be some manual intervention. For instance, the reporting of a specific test case and configuration (below) may require a developer to provide additional information.

If we view a long-running self-adaptive system as executing a sequence of phases, where in each phase a specific configuration is used, then our method is triggered when a failure is detected, advises the reconfiguration process, and then allows the system to run under a new configuration. This process repeats when subsequent failures are detected.

Since we test our hypothesis on a system that is not self-adaptive, we simulate this type of behavior by having a single run of a program's configuration act like a phase in the execution of the long running system. For instance, when compiling a program, one must choose a set of configuration options. If a compilation error, hard crash, or other failure is observed, one might then recompile the same program with a slightly different set of options until the failure is avoided.

Together, the central store and the deployment execute five steps. Step 1 begins the process: whenever a deployment detects a failure, it reports a test case and a configuration to the store. In step 2, an off-line analysis occurs to estimate whether the failure is configuration-dependent, and if so, which configurations it affects. For failures that have reconfiguration workarounds, the analysis results are broadcast in step 3. Step 4 uses this data to forbid configurations believed to

```
1  let $t \leftarrow$ the reported test case;
2  let $c \leftarrow$ the reported configuration;
3  let $d \leftarrow$ the maximum number of feature groups to change at a time (a
   parameter to the technique);
4  let $R \leftarrow$ the set of reconfigurations that affect between one and d feature
   groups;
5  foreach $r \in R$ do
6      let $c' \leftarrow c$ after applying reconfiguration $r$;
7      if $t$ can be run under configuration $c'$ then
8          if $t$ passes under configuration $c'$ then
9              note $r$ as a known workaround;
10             note $c'$ as a passing configuration;
11         end
12     else
13         note $r$ as a possible workaround;
14     end
15 end
16 foreach $r \in R$ do
17     if $r$ is a possible or known workaround whose supersets in $R$ are all either
       possible or known workarounds then
18         note $r$ as a basis for generalization;
19     end
20 end
21 foreach $r \in R$ do
22     if $r$ is a strict superset of a basis for generalization then
23         forget that $r$ is a possible or known workaround;
24         forget that $r$ is a basis for generalization;
25     end
26 end
```

Algorithm 1: Analysis of Failures

be dangerous, both at the time the update is received and when a deployment is reconfigured for other reasons. Lastly, step 5 draws on the historical failure information to suggest workarounds when a new failure appears.

Steps 1 and 3 are simply data transfers. The algorithmic steps 2, 4, and 5 warrant more detail, which we provide in the following subsections.

3.1 Failure Analysis

Step 2, the analysis of reported failures, is listed as Algorithm 1. The main idea is to try, by brute force, configurations that are similar to the one reported (that is, configurations that have many of their features in common) and see which ones pass. Although there are techniques to sample the configuration space more evenly [20], here we want the sampling to concentrate on configurations that will see comparable inputs, presumably those that differ by only a few features.

Lines 1 and 2 begin by establishing the circumstances that led to the original failure, and lines 3 and 4 construct a set of reconfigurations to explore

```
1  let $c \leftarrow$ the current configuration;
2  let $T \leftarrow$ the set of test cases with known workarounds;
3  foreach $t \in T$ do
4      if $c$ is a passing configuration for $t$ then
5          continue with the next iteration of the loop on line 3;
6      end
7      foreach basis for generalization $r$ from $t$ do
8          let $c' \leftarrow c$ after applying reconfiguration $r$;
9          if $c = c'$ then
10             continue with the next iteration of the loop on line 3;
11         end
12     end
13     reject $c$;
14 end
15 accept $c$;
```

Algorithm 2: Guard on Configurations

the vicinity. Throughout these algorithms we treat reconfigurations as sets of operations, each of which overwrite a configuration in one feature group. For example, Spirit would have a reconfiguration that writes “active” to the solar panels’ feature group, even if the solar panels are already in that state. Another reconfiguration might enable both solar panels and the batteries, which, because we think of reconfigurations as sets, would make it a superset.

Each reconfiguration is handled by an iteration of the loop on line 5. If, on line 7, the new configuration that results is both valid and suitable for the test case, the test is run by line 8, and the results are recorded on lines 9 and 10. Otherwise, the algorithm notes that it could not evaluate the reconfiguration, at line 13.

Because the first loop only investigates within a small radius, the technique must make some generalizations to classify the rest of the configuration space. Our experience suggests that while failures may depend on several feature choices, the elimination of any one will usually constitute avoidance. Furthermore, if we are to preserve as much of the intended functionality as possible, we should favor small changes to the system configuration. Therefore, in the absence of contrary evidence, we generalize a workaround reconfiguration by assuming it to always mask the fault, even when other parts of the configuration are radically different.

The second loop, which begins at line 16, is responsible for determining which workarounds can be generalized without contradicting the algorithm’s observations. Per line 17, an effective reconfiguration that is a subset of an ineffective one is not generalized.

Finally, the loop at line 21 discards information that is redundant in light of the generalizations. Specifically, if one reconfiguration is consistently effective, another reconfiguration that makes the same changes plus some extras is not useful; the conditional on lines 22–25 forgets it.

As an example, suppose that Algorithm 1 is applied to a failure that resembles the one Spirit encountered: the system misbehaves when all of four features, f_1–f_4 are present, and, moreover, f_4 is mandated by the test case. First, line 4 will calculate the set R, including a reconfiguration to disable just f_1, a reconfiguration to disable just f_2, etc. For each of f_1–f_3, the corresponding single-feature reconfiguration will cause the test case to pass, so all three will be known workarounds. The reconfiguration eliminating f_4 can't be attempted, so it will be labeled a possible workaround. Any supersets of these reconfigurations must either be impossible to test or also workarounds, so, regardless of d, these four are marked as bases for generalization on line 18. Their strict supersets are subsequently pruned by the loop at line 21, leaving the final diagnosis: any reconfiguration that disables one of f_1–f_3 should avoid the failure; reconfigurations that do not, but do disable f_4, might be effective.

3.2 Configuration Guard

Once the analysis results are available and distributed, Algorithm 2 guards deployments against dangerous configurations. For each test case where Algorithm 1 found feature selection to be significant, the guard checks that the current configuration is either known to be passing, on lines 4–6, or that a general workaround has been applied, on lines 7–12.

Continuing the example from Section 3.1, suppose that a deployment receives notification of the failure caused by f_1–f_4 while in a configuration that enables everything but f_2. Because Algorithm 1 identified the f_2-disabling reconfiguration as effective, and that reconfiguration has no effect on the deployment's current feature choices, it will assume that it does not need to take action. Similarly, if its configuration just disabled f_4, it could also continue, because there is a possibility that a workaround has been applied.

3.3 Choosing New Configurations

The last algorithm, for handling failures that the guard does not avoid, appears as Algorithm 3. Because reconfiguration is potentially expensive, Line 3 establishes a bound on the number of avoidance attempts. The attempts themselves are draw from a pool built on lines 4 and 5, mimicking Algorithm 1, except that reconfigurations are prioritized by their historical effectiveness. The loop on line 7 considers them in order, checking that they are new (on line 9), valid (line 10), and approved by Algorithm 2 (line 11) before making an actual workaround attempt (line 12). The algorithm halts in success if it finds a passing attempt, but in failure if the pool or attempt counter runs out first.

To complete the running example, consider the same deployment encountering a new failure, this time caused by the combination of f_1 and f_5. The pool of historically effective workarounds has reconfigurations to individually disable each of f_1 through f_3, and one of these—the one eliminating f_1—will succeed. In the best case it will come first in the order on line 5 and be tried immediately,

```
let $c \leftarrow$ the current configuration;
let $t \leftarrow$ the current test case;
let $m \leftarrow$ the maximum number of reconfigurations to try (a parameter to the technique);
let $d \leftarrow$ the maximum number of feature groups to change at a time (a parameter to the technique);
let $R \leftarrow$ the set of reconfigurations that affect between one and d feature groups, with elements sorted in decreasing order by the number of test cases they are workarounds for;
let $n \leftarrow m$;
foreach $r \in R$ do
    let $c' \leftarrow c$ after applying reconfiguration $r$;
    if $t$ has not yet been tried under configuration $c'$ then
        if $t$ can be run under configuration $c'$ then
            if Algorithm 2 accepts $c'$ then
                if $t$ passes under configuration $c'$ then
                    accept $c'$;
                end
                let $n \leftarrow n - 1$;
                if $n = 0$ then
                    fail;
                end
            end
        end
    end
end
fail;
```

Algorithm 3: Reconfiguration Strategy for Recovery

but in the worst case the failure won't be avoided until the second try. (Not the third try; f_2 is already disabled.)

3.4 Handling Multiple Versions

Complications arise when a new version of the system is released. The updated system may not have the same set of faults as the old one, and faults that do survive may have different reconfiguration workarounds, especially if the release contains new features.

In our experiments, we track workaround data independently for each version. At release, the central store checks for all of the known failures and collects suitable reconfigurations for those that are found. Furthermore, when a failure is detected in one version, the other active versions are checked, if possible, under the same test case.

Under this policy, forbidden feature combinations become available again as soon as the known faults are fixed. However, research also shows that new faults tend to appear in places where old ones were found [26]. It might be worthwhile to continue using data from old failures, especially if faults are fixed quickly, and so reduce the risk of failure at the expense of functionality. We plan to investigate this tradeoff in future work.

4 Case Study

As an initial evaluation of how well reconfiguration can be used to avoid failures, we conducted a case study with 128 failures reported in the field for at least one of three versions of a highly-configurable software system. We repeated this study on the same software system using test cases randomly generated by a state of the art testing tool, Csmith; these invoked a different set of failures.

The study's research questions are presented in Section 4.1, and the systems, GCC is covered by Section 4.2. We describe our experimental methodology in Section 4.3, our threats to validity in Section 4.4, and we discuss the results in Section 4.5. Experimental results can be found at `http://www.cse.unl.edu/~myra/artifacts/locality` .

4.1 Research Questions

For our technique to be useful, there must be failures that it can work around. Hence, we first asked,

RQ1: Can failures be avoided by reasonable reconfigurations?

Provided that such failures exist, we must determine whether they exhibit feature locality:

RQ2: To what extent do failures depend on similar combinations of features?

If the failures are present and localized, then we can ask about the effectiveness of our technique:

RQ3: Can feature locality be exploited to avoid failures?

And finally, if we can avoid failures, we should ask whether we are only doing so temporarily, for the use case at hand, or more generally, for that and subsequent use cases:

RQ4: How robust are reconfiguration workarounds against use case changes?

4.2 Objects of Study

We evaluated our proposed technique on several versions of one highly-configurable software system, GCC.[1] Although there are characteristics of

[1] We used GCC (and another application) and a nearly identical initial failure pool in separate publication [39]. The only overlap with this work is that [39] employed Algorithm 1 to determine whether failure were configuration dependent. Beyond that, we performed a manual study of the faults at the code level for a different purpose.

self-adaptive systems that it cannot capture, it is very representative of highly-configurable software and has some characteristics (described below) that we believe allow us to simulate such a system. Also, GCC has an active user community and a public bug database, which we can mine for failures [40].

As in a distributed self-adaptive system, GCC deployments process separate streams of inputs (test cases), namely sequences of compilation tasks including those in the bug reports, and operate under changing configurations (command-line options) for which we can extract a timeline from the bug database. Likewise, they are free to exchange failure information, here via human-generated reports for the GCC bug database (our central store). Because instances of the compiler are usually isolated from each other, we do not consider effects due to node interactions. Moreover, GCC reconfigurations can only happen between runs, so our evaluation does not capture behavioral changes during reconfiguration.

The following subsections describes this system in more detail, with an emphasis on how we obtained the feature model, failures, initial configurations, and so on.

GCC Versions. GCC [41], the cornerstone of the GNU toolchain, is a compilation framework with front-ends for a variety of languages and back-ends for a variety of platforms. The case study covers versions 4.4.0–4.4.2, all released in 2009, which each exceed 23 million lines of code.

In constructing GCC's feature model, we restricted ourselves to the compiler's command-line options, grouping features according the manual. We only included features that can be toggled without changing the input or the semantics of the output. One case deserves special explanation: GCC has some features that cannot be controlled completely from the command line. For example, the standard optimization packages (`-O1`, `-O2`, `-Os`, and `-O3`) enable some optimizations that have no corresponding flag. To handle these otherwise inaccessible behaviors we treated the use of each package as a feature, and put these pseudo-features in one group. If a failure depends on a hidden optimization from `-O2`, the technique will suggest workarounds like switching to `-O1` and listing the lost optimization flags explicitly.

We also assumed that several options are dictated by the test case being run: the stages of compilation to execute, the input language and its extensions (even when those extensions were not used), the platform or platforms being compiled for, the application binary interface, and the debug information that is emitted. This information is used by line 7 of Algorithm 1 and when selecting the new configuration to ensure that workarounds preserve the user's intended behavior.

The complete model, including pseudo-features, totals 321 features in 159 groups, which means 162 single-feature reconfigurations are possible from any one starting point. All but one of the groups is binary (they each have two alternatives), while one has five possible choices. We also enumerated the dependencies among our features. In total we have 132 clauses to represent these constraints on the combinations of features in GCC. For example, if we turn on the feature `-fsched-spec-load`, speculative motion of load instructions, then according to the GCC documentation we should also run instruction rescheduling before register allocation by enabling `-fschedule-insns`, `O2`, or `O3`.

User-Reported Failures. For one set of failures, we collected 360 reports from GCC's public bug database [40] that affect compilation or debugging for C, C++, and Fortran programs and are also tagged with "known to fail" on at least one of the versions in the 4.4.0–4.4.2 range. We chose an appropriate subset for the experiments:

First, we removed seven of these reports because they were still incomplete.

Then we discarded another 92 that depend on the platform where GCC is built, the platform where it runs, or the platform that it compiles for. Although we could have easily included failures that affect our platform—a 64-bit X86 system running openSuSE 11.0, which the auto-configuration detected as `x86_64-unknown-linux-gnu`—and we also could have used simulators to reproduce failures that call for other platforms, we were aiming to make our case study portably reproducible.

Next, because the bootstrap process that builds GCC is itself configurable, we further excluded three failures that required a non-default bootstrap configuration.

Finally, we omitted two other classes of failures: those where the problem is a violation of a nonfunctional requirement so we could not obtain an indisputable oracle (13 failures), and those that showed nondeterministic behavior (8 failures).

In summary, of the original 360 failures we kept nearly two thirds, 237. A synopsis of the failures excluded for various reasons is given in Table 1.

Table 1. User-Reported Failures

	GCC	
Reported	**360**	**(100.0%)**
Incomplete	7	(1.9%)
Platform-Dependent	92	(25.6%)
Require Alternate Bootstrap Options	3	(0.8%)
Nonfunctional	13	(3.6%)
Nondeterministic	8	(2.2%)
Remaining	**237**	**(65.8%)**
Reproducible on Releases	**128**	**(35.6%)**

We then checked for each failure under every GCC version in our study. Almost half of the remaining failures were only visible in pre- or post-release revisions, so we could not reproduce them with the released code. On the other hand, of the failures we could reproduce, most affected all three versions despite being tagged with only one of the three as known-to-fail. The release of 4.4.1 showed only three failures that were not in 4.4.0; only two more were added from 4.4.1

to 4.4.2. The total line of Figure 2 shows the numbers for each version; the remainder of this table will be discussed in Section 4.5.

Finally, we used time stamps on the bug reports and the history of releases in the GCC SVN repository to build an overall picture of the sequence of events.

Csmith-Detected Failures. For our second failure set, we used the Csmith random test generator [28] in its default configuration. Csmith creates programs subject to the constraint that they be legal and have only one interpretation under the C99 standard [42]. This was essential for our study—it would be far too easy to provoke false failures otherwise. At the end of their execution, these programs hash their state and print the result. Rather than manually develop oracles for these test cases, we compared against a recent version of GCC. If a program compiled (meaning that the compiler did not crash), ran to completion (meaning that the emitted binary did not crash), and output a hash that matched the program compiled under GCC 4.6.2, we declared the test case to pass; otherwise we considered it to fail. After creating 1024 test cases, we ran them under the same versions of GCC for which we had collected user-reported failures from the field: 4.4.0–4.4.2. As for starting configurations, we used the flag sets that the GCC torture tests apply (see below), which, apart from the empty set, all led to the same 16 of the 1024 test cases failing. There were no failures when no command-line options were given.

Torture Test Suites. The GCC test suites contains a set of tests, called torture tests, that are designed to run under various configurations. These configurations are specified in the test harness, and we used them for the Csmith tests, as explained above. But we also employed the test cases in other configurations for our fourth research question; like the Csmith test cases, we could depend on these tests' oracles to hold in the face of reconfiguration.

When counting torture tests, we used the pass/fail totals reported by the DejaGNU harness. In particular, the counts do not include expected failures, unexpected passes, or test cases included with GCC but not run by the harness.

4.3 Methodology

For each failure, we are interested in the number of reconfigurations that avoid it and how many tries Algorithm 3 needs to choose such a reconfiguration, in the best case and in the worst case. For each reconfiguration, we want to determine the number of failures it avoids. Because the association between failures and reconfigurations is already determined in Algorithm 1, we collect all of this data by simulating our technique.

The simulation must consider two types of events: the release of a new version and the discovery of a failure. It accounts for a release by running Algorithm 1 on every failing test case that is marked as seen before. To process a failure the simulation must follow a more complicated procedure.

First it loops through the versions that are deployed and applies Algorithm 1. Whenever the failure can only be achieved in configurations that Algorithm 2 would reject, the simulation notes that our technique would avoid the failure with zero reconfiguration attempts. Otherwise the failure is marked as seen.

For efficiency's sake the study only considers reconfiguration workarounds that change a single feature group; we set d in Algorithms 1 and 3 to one. As a consequence, the simulation may wrongly classify failures as unavoidable and bias the data against our technique. We also reran the experiment with d set to two, but the data remained identical. However, we cannot make conclusions beyond $d = 2$.

Next, for those versions where our technique cannot avoid the failure outright, the simulation determines the number of tries that the Algorithm 3 would need to suggest a failure-avoiding reconfiguration. Ties in the sorting of reconfiguration attempts are broken to favor ineffective alternatives when we compute the worst case and effective choices in the best case.

Finally, if the failure was seen in any version, the results of Algorithm 1 are saved. But if no version saw the failure, no data is kept for future use. There are two key assumptions here: First, we expect that the feature isolation process can be completed before the next failure is discovered. For GCC this requires only a few minutes—much less than the typical interval between bug reports. Second, we pessimistically permit multiple versions to encounter the same failure simultaneously.

Biased Random Reconfiguration. For comparison, we also simulated a technique that does not exploit the information gathered by the central store, but that uses other knowledge of the system which our technique does not have. It is meant to represent the approach to failure avoidance by reconfiguration that has an experienced user intervene when a failure occurs. First, the configuration space is not pruned, so every failure will be seen. Second, the attempted reconfigurations are chosen randomly, but with a bias towards workarounds that will succeed. The bias encodes the advantages of considering the type of failure, the input that triggered it, white-box knowledge, etc. We simplify the model by assuming that all ineffective reconfigurations have the same probability p, and that all effective choices have some probability q.

Because there is an element of randomness in this alternative technique, its worst case is to try the viable workarounds last and its best case to try them first. For the sake of a meaningful comparison, we consider its average case.

Let R be the set of candidate reconfigurations, $R^+ \subseteq R$ be the set of reconfigurations that will prevent a failure, and R^- be $R \setminus R^+$. If we assume—to the disadvantage of our approach—that the competing technique is not hampered by some configurations being illegal, the probability of it avoiding that failure within r reconfigurations is:

$$1 - \binom{|R^-|}{r} \Big/ \binom{|R^-| + \frac{q}{p}\,|R^+|}{r}. \tag{1}$$

The calculation applies to exactly one failure, so the number of failures avoided in the average case is equal to this probability. Hence, for each test case we add the result of (1) to the avoidance count for the competing approach.

Robustness Analysis. To assess the workarounds' robustness against changing use cases, we ran several thousand test cases—those from user-reported faults, those generated by Csmith, and those in the GCC torture test suite of the same version—under the configurations suggested by each successful workaround. We also counted the number of failures that could be seen after applying the workaround to some other initial configuration, under the assumption that the features governing a fault are exactly those observed by Algorithm 1.

4.4 Threats to Validity

The major threat to the external validity of our study is the fact that we consider a single system, which clearly differs from a real self-adaptive system in certain aspects—whether those aspects are critical to its reconfigurability is not known. Although this system had a significant share of reconfiguration-avoidable failures, and these failures exhibited feature locality, further work is necessary to understand if these properties hold for self-adaptive software in general.

However, we have several reasons to believe that the results will generalize. Prior work in combinatorial testing [23, 39] suggests that the dependencies we observed between failures and features are typical of many systems, not just compilers. Similarly, almost all of the reconfiguration-avoidable failures that we can trace are caused by faults in specifically feature-dependent code [39]. As we can expect such code to appear in a variety of configurable systems [32], and we know that spacial locality of faults occurs in a diversity of systems [25, 26], we can expect at least one likely cause for feature locality of failures to persist in other settings.

Another threat is the possibility that we only observed locality patterns due to the way the failures were uncovered. For user-reported failures, it might be that users who report bugs tend to use the compiler in similar ways. In the case of Csmith, although the tool draws test cases from a random distribution, this distribution is far narrower than the set of all inputs that GCC might reasonably encounter. Moreover, we do not know whether the 16 failures from Csmith tests are actually due to 16 distinct faults. If not, the shared faults might make locality appear stronger than it is. To partially mitigate this threat, we included both kinds of failures in our study and note that they exhibited similar trends.

In our evaluation of robustness, we again have to approximate a reasonable range of GCC inputs. Therefore, besides the test cases from user-reported failures and Csmith, we also included the GCC torture tests. However, the input space for GCC is large enough that our test suite might not be representative.

Within our analysis of field failures broken down by priority, we have a small number of faults in the highest priority category. This may limit our ability to make general conclusions.

Last, we used a recent version of GCC as the oracle for the Csmith test cases when neither the compiler nor the emitted binary crashed. It is possible that these tests also failed under the recent version, although the fact that we could match its behavior by omitting optimization flags suggests otherwise.

For internal validity, we must acknowledge the risks in the manual categorization and encoding of bug reports, the limitations of having only one test case to provoke each failure, and the ever-present risk of faults in our evaluation code.

Regarding construct validity, there is a chance that the reconfiguration workarounds we propose may cause systems to encounter failures that are not in the bug database; in that case, our reported rate of success will not be achieved.

4.5 Results

In the following subsections we discuss the results of our simulation in the context of each research question.

RQ1: Can Failures be Avoided by Reconfiguration? For our first research question, we only considered user-reported failures, because our aim was to understand how often reconfiguration can be used for workarounds in the field. The ability to avoid other failures is only relevant if the other failures are representative of those that deployments will encounter, which we do have enough data to assess.

Table 2 presents the reconfiguration workarounds we discovered for user-reported failures, organized by failure and version. Each row presents the number of one-step reconfigurations that are possible. The first row shows the number of failures that had no workarounds at all, while failures with suitable configuration dependence are counted in subsequent rows. At the bottom we total the failures in each version and give the portion that have at least one known workaround, both as a count and as a percent of the total. For instance, in GCC 4.4.0 there are 99 failures with no workarounds and 27 that have at least one. If we examine the sixth row, we see that there are three failures that have five one-step workarounds. This means that for each failure we can toggle any one of five features and the failure will no longer occur.

Roughly one fifth of the failures in each version are sensitive to reasonable reconfigurations, so the choice of features does play a significant role in a system's reliability. We also observe that, as in the study of Kuhn et al. [23], most of these failures are affected by only a few features—usually no more than six or seven out of 321 (roughly 2%). But we do see a handful of exceptions that can be avoided by changing any one of 8–13 different features. In these cases, there happens be a long data flow chain that invokes that failure, and breaking it at any point prevents the failure's occurrence.

We next performed an analysis to see if high-priority failures have different characteristics than those with lower priorities. We used the rankings given by GCC developers to focus their fault-fixing efforts: from P1 (the most urgent) to P5 (unimportant).

Table 2. User-Reported Failures with One-Step Workarounds

# of One-Step Workarounds	**Counts** GCC 4.4.0	GCC 4.4.1	GCC 4.4.2
0	99	91	85
1	5	5	4
2	5	4	3
3	4	4	3
4	4	3	3
5	3	3	3
6	2	2	1
7	2	2	2
8		1	1
9		1	
10		1	
11	1		
12			
13	1		
Total	126	117	105
Nonzero	27	26	20
Percent Nonzero	21%	22%	19%

Table 3. User-Reported Failures in GCC with Reconfiguration Workarounds, by Priority

	4.4.0	**4.4.1**	**4.4.2**
P1	3 of 5 (60%)	3 of 4 (75%)	2 of 3 (67%)
P2	6 of 23 (26%)	6 of 19 (32%)	4 of 17 (24%)
P3	18 of 84 (21%)	17 of 80 (21%)	14 of 75 (19%)
P4	0 of 11 (0%)	0 of 11 (0%)	0 of 8 (0%)
P5	0 of 3 (0%)	0 of 3 (0%)	0 of 2 (0%)
Total	27 of 126 (21%)	26 of 117 (22%)	20 of 105 (20%)

Table 3 shows the breakdown. We see that none of the low-priority (P4 and P5) failures have reconfiguration workarounds, but a majority of P1 failures do, which is encouraging. Percentage-wise we see a trend where failures with workarounds are more likely to be promoted from the default P3 status, meaning that the failures that matter most to developers are also avoidable by reconfiguration.

Summary of RQ1. In summary, approximately one fifth of the failures can be avoided by reconfigurations, with the fraction being larger for high-priority bugs (60–75% for GCC P1 reports).

RQ2: To what Extent do Failures Depend on Similar Combinations of Features? Next, we view the same data from the perspective of reconfigurations to determine whether failures share workarounds. For each single-feature reconfiguration, we counted the number of bug reports where the reconfiguration avoided that failure. After sorting counts from highest to lowest, we plotted

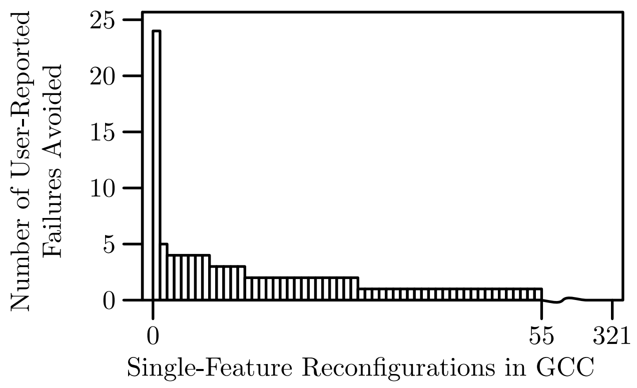


Fig. 2. Feature Locality of User-Reported Failures

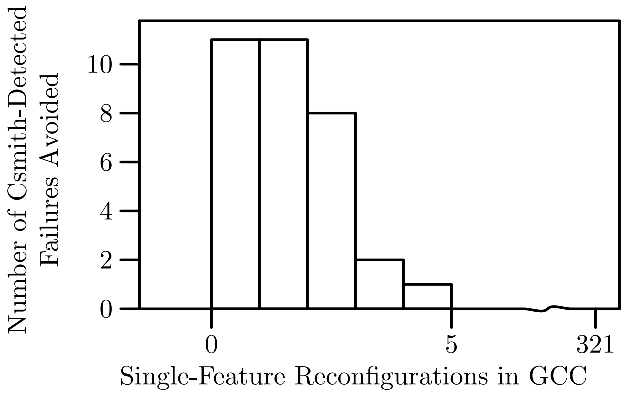


Fig. 3. Feature Locality of Csmith-Detected Failures

them in Figures 2 (user-reported) and 3 (Csmith-detected). Note the broken scales on the x-axes, which elides 266 bars of height zero in Figure 2 and 316 in Figure 3. This indicates that 260 features had no impact on avoiding any failure. The spike on the far left of Figure 2 corresponds to disabling undocumented optimizations by lowering the optimization level and then re-enabling the documented optimizations associated with the old level. This single reconfiguration avoids 24 failures on its own, but at the cost of the undocumented functionality.

We do see feature locality in the graphs: less than 20% of the reconfigurations appear to affect correctness in the user-reported failures, and the percentage drops to 2% for the Csmith-detected failures. In an autonomic setting, most of the feature choices would be free to vary in response to other concerns. Furthermore, the height of the bars shows that failures tend to have overlapping workarounds, and avoiding one failure often means avoiding others.

The same data is shown in expanded form in Tables 4 and 5. The rows list failing bug report and version pairs, grouped by report number, while the

Table 4. Workarounds Effective Against User-Reported Failures

Bug	GCC	-W	-Wextra	-fbranch-probabilities	-fno-caller-saves	-fno-dse	-fno-early-inlining	-ffloat-store	-fno-gcse*	-fgcse*	-finline-functions	-fno-inline-functions-called-once	-fno-inline-small-functions	-fno-ipa-cp	-fno-ipa-cp-clone*	-fipa-cp-clone*	-fno-ivopts	-fno-move-loop-invariants	-fnon-call-exceptions	-fno-optimize-register-move	-fno-optimize-sibling-calls	-fpack-struct	-fpeel-loops	-fpredictive-commoning	-freg-struct-return	-fno-regmove	-frerun-cse-after-loop	-fschedule-insns	-fno-strict-aliasing	-fno-toplevel-reorder	-ftrapv	-fno-tree-ccp	-fno-tree-ch	-fno-tree-copy-prop	-ftree-dce	-fno-tree-dominator-opts	-fno-tree-fre	-fno-tree-loop-ivcanon	-fno-tree-loop-optimize	-fno-tree-pre	-fno-tree-sra	-fno-tree-ter	-fno-tree-vectorize	-fno-tree-vrp	-fno-unit-at-a-time	-funroll-all-loops	-fno-unroll-loops*	-funroll-loops*	-fno-unsafe-math-optimizations	-fwrapv	<omit -fno-ira-share-save-slots>	<choose undocumented -O0 features>*	<choose undocumented -Os features>*	<choose undocumented -O1 features>*	<choose undocumented -O2 features>*	<choose undocumented -O3 features>*
25689	4.4.0	●	●																																																					
25689	4.4.1	●	●																																																					
25689	4.4.2	●	●																																																					
31029	4.4.0																●																						●																	
31029	4.4.1																●																						●																	
31029	4.4.2																●																						●																	
36902	4.4.0																															●												●								●				
36902	4.4.1																															●												●								●				
36902	4.4.2																															●												●								●				
38540	4.4.0			●																																																				
38540	4.4.1			●																																																				
38540	4.4.2			●																																																				
38541	4.4.0			●																																																				
38541	4.4.1			●																																																				
38541	4.4.2			●																																																				
38808	4.4.0						●						●						●																		●				●			●								●				
38808	4.4.1						●						●						●																		●				●			●								●				
38808	4.4.2						●						●						●																		●				●			●								●				
39794	4.4.0					●					●					●	●		●					●				●			●								●			●					●					●				
40321	4.4.0																		●		●																			●	●											●				
40321	4.4.1																		●		●																			●	●											●				
40389	4.4.0																																								●											●				
40579	4.4.0																●						●								●								●					●		●		●		●		●		●		●
40924	4.4.1					●			●									●														●								●		●										●	●				●
41016	4.4.0																																																			●				
41016	4.4.1																																																			●				
41094	4.4.0																																																●							
41094	4.4.1																																																●							
41153	4.4.0																								●																											●				
41153	4.4.1																								●																											●				
41153	4.4.2																								●																											●				
41183	4.4.0										●			●																●															●							●				
41183	4.4.1										●			●																●															●							●				
41183	4.4.2										●			●																●															●							●				
41287	4.4.1					●				●								●					●				●															●				●		●				●			●	
41403	4.4.0																																																			●				
41403	4.4.1																																																			●				
41403	4.4.2																																																			●				
41619	4.4.1				●			●			●								●	●						●																									●	●				
41619	4.4.2				●			●			●								●	●						●																									●	●				
41623	4.4.0																																																				●	●	●	●
41630	4.4.0											●																													●											●				
41630	4.4.1											●																													●											●				
41630	4.4.2											●																													●											●				
41643	4.4.0																				●													●	●																	●				
41643	4.4.1																				●													●	●																	●				
41643	4.4.2																				●													●	●																	●				
41646	4.4.0					●																●																														●				
41646	4.4.1					●																●																														●				
41917	4.4.0																																																			●				
41917	4.4.1																																																			●				
41917	4.4.2																																																			●				
42049	4.4.0																															●	●				●						●		●								●				
42049	4.4.1																															●	●				●						●		●								●				
42049	4.4.2																															●	●				●						●		●								●				
42231	4.4.0						●						●	●	●																																					●				
42231	4.4.1						●						●	●	●																																					●				
42231	4.4.2						●						●	●	●																																					●				
42488	4.4.0																												●			●					●				●											●				
42488	4.4.1																												●			●					●				●											●				
42488	4.4.2																												●			●					●				●											●				
42542	4.4.0																																						●				●									●				
42542	4.4.1																																						●				●									●				
42542	4.4.2																																						●				●									●				
42614	4.4.0																												●									●														●				
42614	4.4.1																												●									●														●				
42614	4.4.2																												●									●														●				
42667	4.4.0																																							●												●				
42667	4.4.1																																							●												●				
42667	4.4.2																																							●												●				
43024	4.4.0													●																●															●							●				
43024	4.4.1													●																●															●							●				
43024	4.4.2													●																●															●							●				

columns are labeled with single-feature workarounds, usually a flag to pass like `-fno-caller-saves`, or, when written in angle brackets, as in `<omit -fno-ira-share-save-slots>`, a short description of how the command-line flags should be altered. They are grouped in blocks of five in Figure 4 to allow one to read the table more easily; the groupings have no other meaning. At the intersection of a row and column, a dot appears if the workaround was found effective against that particular failure. As in the previous figures, we do not show those features that do not provide workarounds.

Examining Figure 4, we first see the column of dots for `-O0` that corresponds to the spike in Figure 2. The table also identifies the other features that cross multiple reports (e.g., `-fno-tree-sra`), and those that workaround only a single report's failures (such as `-W`). Most of the reports are pairwise dissimilar; it is only in aggregate that trends start to emerge.

Also noteworthy are some conflicts in the reconfigurations that failures might suggest. In particular, bug 41623 contradicts the common trend that failures can be avoided by specifying `-O0` and writing the lost optimization options explicitly. Other conflicts occur in global common subexpression elimination (`gcse`), function cloning to strengthen interprocedural constant propagation (`ipa-cp-clone`), and loop unrolling (`unroll-loops`), whose columns are marked with asterisks. None of these would pose a serious challenge to our failure avoidance technique, because all of the affected bugs have alternative reconfiguration workarounds. However, the disagreements do matter when we evaluate robustness for RQ4.

Turning to the Csmith detected failures, (Figure 5), two facts immediately stand out: there are far fewer columns, and all but one of the workarounds is effective against multiple failures. Thus, we see much stronger locality, even though Csmith is an input-based, not a feature-based testing tool. That further suggests a connection between inputs and the reconfigurations that are likely to be effective. We leave this analysis for future work.

The Csmith data does conflict with the user-reported failures on one point: dead code elimination (`tree-dce`) masks bug 41643, but Csmith bug 528 disappears when it is disabled. This conflict resurfaces in RQ4.

Summary of RQ2. Our data shows strong feature locality: a few reconfigurations have a significant impact on failure visibility, while we detected no effect for the remainder. Accordingly, in an autonomic setting, we need only consider a small number of features when predicting and reconfiguring for functional correctness.

RQ3: Can Feature Locality be Exploited to Avoid Failures? To answer RQ3, we compiled the simulation results to see if our technique was in fact effective. Figure 4 shows the data for the three versions of GCC on the left for user reported failures. The limit on the number of reconfiguration tries, up to a limit of 25, is on the x-axis of each plot, and the number of faults avoided is on the y-axis. We shade the region between our technique's best and worst cases; its performance must fall in this region. For comparison, we also show the

Table 5. Workarounds Effective Against Csmith-Detected Failures

Bug	GCC	-fno-ipa-reference	-fno-tree-dce	-fno-tree-fre	<choose undocumented -O0 features>	<choose undocumented -O3 features>
045	4.4.0			●	●	
045	4.4.1			●	●	
045	4.4.2			●	●	
110	4.4.0	●		●	●	
110	4.4.1	●		●	●	
110	4.4.2	●		●	●	
112	4.4.0	●		●	●	●
112	4.4.1	●		●	●	●
112	4.4.2	●		●	●	●
158	4.4.0	●		●	●	
158	4.4.1	●		●	●	
158	4.4.2	●		●	●	
201	4.4.0	●		●	●	
201	4.4.1	●		●	●	
201	4.4.2	●		●	●	
203	4.4.0			●	●	
203	4.4.1			●	●	
203	4.4.2			●	●	
299	4.4.0	●		●	●	
299	4.4.1	●		●	●	
299	4.4.2	●		●	●	
514	4.4.0			●	●	●
514	4.4.1			●	●	●
514	4.4.2			●	●	●
528	4.4.0	●	●	●	●	
528	4.4.1	●	●	●	●	
528	4.4.2	●	●	●	●	
888	4.4.0	●		●	●	
888	4.4.1	●		●	●	
888	4.4.2	●		●	●	
983	4.4.0	●		●	●	
983	4.4.1	●		●	●	
983	4.4.2	●		●	●	

average case for random reconfiguration with various degrees of bias. The lowermost line is the expected behavior when effective and ineffective reconfigurations are equally probable, the next line makes workarounds twice as likely, and so on, until the topmost line, where failure-avoiding choices are preferred 64 to one.

For instance, on GCC 4.4.1, our technique avoids either 19 or 20 failures within three reconfigurations, slightly better than we would expect from biased randomly chosen reconfigurations when the effective choices are 64 times more likely to be picked. In every case, the technique prevents more than half of the reconfiguration-avoidable GCC failures from ever being seen, and the proportion increases in later versions because we retain information about surviving failures between versions. Using biased random reconfiguration as a ruler, our technique is more than four times as likely to choose correctly after even 25 reconfigurations, and its performance matches much higher levels of bias earlier on. In short, GCC's feature locality makes historical workarounds good candidates for newly encountered failures.

We show another view of this data on the right-hand side of Figure 4. The x-axis lists each of the 35 bug reports corresponding to reconfiguration-avoidable failures in chronological order. The intervals plotted against the y-axis give the best- and worst-case number of reconfiguration attempts needed to avoid each failure, with an interval omitted when the corresponding failure does not affect that GCC version. Note the break in the y-axis. We had no prior information for some reconfiguration-avoidable failures, and in these cases our technique could do no better than guess, which means at worst that it will try all 162 possible changes.

The main trend is captured in the plot for GCC 4.4.0: after an initial burst of learning from four failures that the technique is unable to avoid, the accumulated knowledge prevents nearly three out of every four failures outright, with all others avoidable in a handful of attempts. Meanwhile, in version 4.4.1, the patterns gleaned from 4.4.0 speed up the avoidance process, and in 4.4.2 everything is prevented. Across all versions, 78% of failures are sidestepped without the technique resorting to guessing reconfigurations.

Figure 5 shows a plot for the Csmith data. Because these failures are unordered, we chose the order under which our technique performs most poorly. Also, because neither releases nor fixes are interleaved with these failures, we pessimistically assume that no information is retained from version to version. Therefore, the plot is the same for all three, and we only display it once.

Like Figure 4, Figure 5 shows most failures being completely avoided, and others circumvented within a few reconfigurations, despite the antagonistic failure ordering. So the same conclusions apply to these failures as to those reported by users.

Summary of RQ3. Our simulation suggests that, because of the feature locality of failures, a system's failure history provides guidance for reconfiguration that effectively avoids new problems.

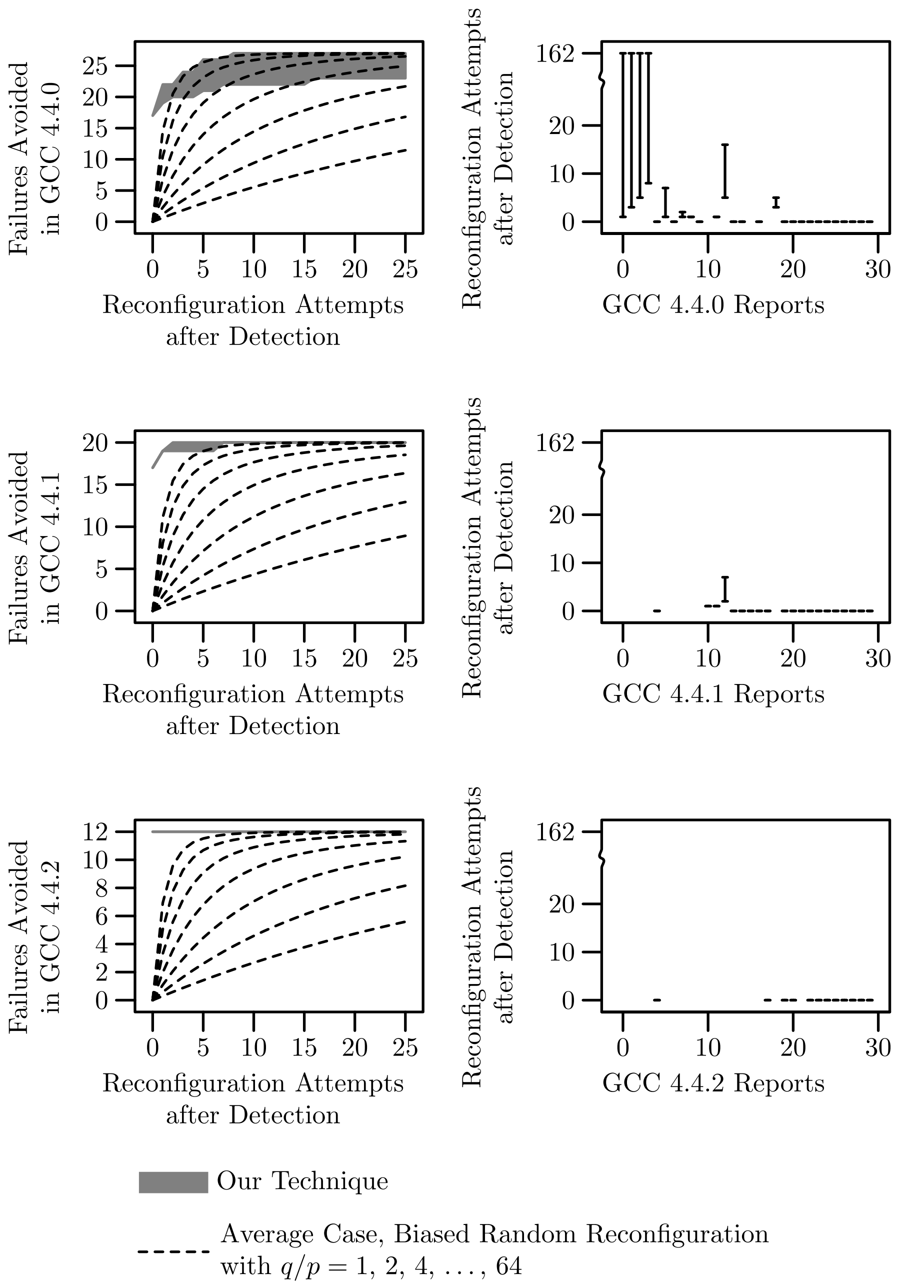


Fig. 4. Number of User-Reported Failures Avoided versus the Number of Reconfigurations after Detection

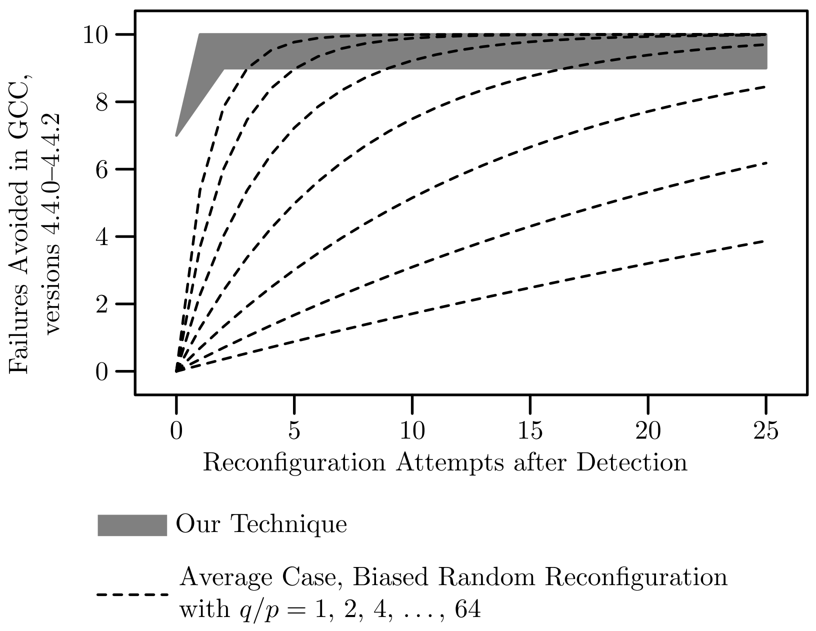


Fig. 5. Number of Csmith-Detected Failures Avoided versus the Number of Reconfigurations after Detection

RQ4: How robust are reconfiguration workarounds against use case changes? For our fourth question, we considered pairings of reconfiguration-avoidable failures with test cases, asking whether it would be possible for each test case to fail in a configuration suggested by each failure. If it often happens that reconfiguring to avoid one failure puts the system in a situation where others can appear, then the workarounds lack generality, and mostly serve to tradeoff different kinds of broken behavior. But, if not, as we would expect from feature locality, then they are useful reconfigurations to carry forward.

We found that the 1258 torture tests did not detect any failures after any of our reconfigurations, nor did the 1024 Csmith test cases after reconfigurations for Csmith-detected failures. However, there was one user-reported failure that could lead to a Csmith test case failing and another where 11 Csmith tests could detect post-reconfiguration trouble—these were due to the conflicts over dead code elimination and the undocumented `-O0` features mentioned earlier. The 128 test cases from the user-reported failures also turned up a number of failures after reconfiguration: 12 total after working around 11 Csmith-detected failures, and 48 after avoiding 21 user-reported problems. Again, these all traced back to the conflicts mentioned under RQ2.

The summary of our findings is given by Figure 6, which includes both user-reported and Csmith-detected failures. Along the x-axis we plot failures with reconfiguration workarounds sorted from most able to reconfigure into a failure to least able. The y-axis shows the number of possible failures, with an upper

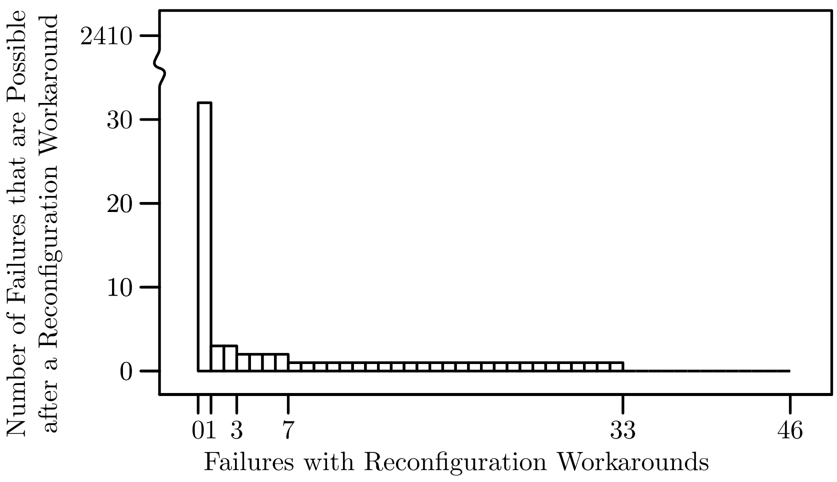


Fig. 6. Robustness of Reconfiguration Workarounds, Numerically

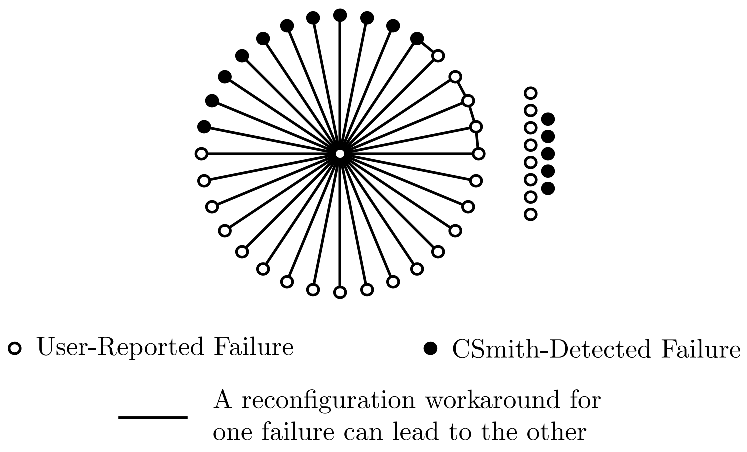

○ User-Reported Failure
● CSmith-Detected Failure
A reconfiguration workaround for one failure can lead to the other

Fig. 7. Robustness of Reconfiguration Workarounds, Graphically

bound at 2410 for the total number of test cases. The leftmost bar, for instance, which reaches 1.3% of the total test cases, represents GCC bug 41623, whose workarounds increase the optimization level and potentially provoke any one of the bugs that disappear under `-O0`. It also accounts for one unit of height in the 33 non-zero columns—it turns out that every failure it could reconfigure into could also reconfigure into it.

However, if we ignore that atypical bug, the results look quite promising. There are only six other bugs where we risk a post-reconfiguration failure under the three test suites. They can be seen more clearly if we view the data graphically, as in Figure 7.

In that figure, open dots (including the one in the center) represent user-reported failures, while closed dots are for those that Csmith detected. Edges show pairs where avoiding one failure could cause the other's test case to fail and vice-versa. The relation is symmetric, so we do not show directions on the edges, and it did not involve any torture tests, so no vertices are included for them.

Bug 41623 appears at the center of the pinwheel, and, as we remarked earlier, without it, most failure's vertices would be completely disconnected. The six exceptions are the dots on the upper-right of the wheel, mostly user-reported. Each of these might reconfigure to one of their peers, depending on the reconfiguration that we choose. The longest possible walk through these failures is four reconfigurations long.

We should keep in mind, however, that the average case is much better than this worst case. First, Algorithm 2 would prevent many of these failures from ever occurring. Second, the failures that could manifest after a reconfiguration all depend on several features—all of them but the one changed by the reconfiguration would have to be appropriately set beforehand. Third, many of these failures are highly sensitive to the compiler's input and would be hard to provoke even after a reconfiguration introduces a suitable configuration.

Summary of RQ4. The reconfigurations suggested by the technique are highly robust, except in cases where bug 41623 is involved. Avoiding this bug's failures might well lead to worse problems, and it may occasionally appear when we avoid others. But apart from 41623, we could only detect potential failures after six of the reconfigurations (13%). Even when possible, these failures affected few test cases; even the bar on the left of Figure 6 represents only 1.3% of the combined suite.

5 Conclusions

Building on prior work towards self-adaptation for correctness, we proposed a framework for failure avoidance by reconfiguration. Rather than modeling failures in aggregate, our framework models individual failures' dependence on the system configuration, as these models can be learned more quickly and with less effort. It then exploits feature locality, a tendency for failures to depend on similar combinations of features, to predict future failures' behavior according to historic failure models.

In our case study, we find that the technique performs quite well preventing and reconfiguring away from those failures that it targets. Based on data from a widely-used highly configurable system, about one in five failures from the field are reconfiguration-avoidable, so there is plenty of ground on which to apply our approach. Among these failures we detect some evidence of feature locality, especially if we concentrate on failures on similar inputs. Therefore, our algorithms avoid almost all avoidable failures in a handful of reconfiguration attempts, and, by the system's third version, prevent all such failures completely.

The reconfigurations also prove fairly robust: apart from one anomaly, they almost never lead to additional failures.

The work suggests several lines of future work. We plan to locate suitable self-adaptive systems, extend the study to their failure histories, and incorporate our algorithms while accounting for tradeoffs between correctness and other concerns. It would also be worthwhile to consider failure-clustering techniques, as suggested by the Csmith data, investigate why reconfiguration-avoidable failures are more likely to trace to high-priority faults, and evaluate various strategies for remembering workarounds even after the associated failures have been fixed.

Acknowledgments. This work is supported in part by the National Science Foundation through awards CNS-0720654, CCF-0747009, CCF-0915526, and CFDA-47.076, by the Air Force Office of Scientific Research through awards FA9550-09-1-0129, FA9550-09-1-0687 and FA9550-10-1-0406. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the position or policy of NSF or the AFOSR.

References

1. Carzaniga, A., Gorla, A., Pezzè, M.: Self-healing by means of automatic workarounds. In: International Workshop on Software Engineering for Adaptive and Self-managing Systems, pp. 17–24 (2008)
2. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: Towards architecture-based self-healing systems. In: Proceedings of the First Workshop on Self-healing Systems, pp. 21–26 (2002)
3. Denaro, G., Pezzè, M., Tosi, D.: Ensuring interoperable service-oriented systems through engineered self-healing. In: Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 253–262 (2009)
4. Chang, H., Mariani, L., Pezze, M.: In-field healing of integration problems with COTS components. In: Proceedings of the International Conference on Software Engineering, pp. 166–176 (2009)
5. Gomaa, H., Hussein, M.: Model-based software design and adaptation. In: International Workshop on Software Engineering for Adaptive and Self-Managing Systems, p. 7 (2007)
6. Brun, Y., Medvidovic, N.: Fault and adversary tolerance as an emergent property of distributed systems' software architectures. In: Proceedings of the Workshop on Engineering Fault Tolerant Systems, p. 7 (2007)
7. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: International Conference on Software Engineering, pp. 371–380 (2006)
8. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems 4(2), 1–42 (2009)
9. Mahadevan, N., Dubey, A., Karsai, G.: Application of software health management techniques. In: Proceedings of the Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS, pp. 1–10 (2011)
10. Pullum, L.: Software Fault Tolerance: Techniques and Implementation. Artech House, Inc. (2001)

11. Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. 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)
12. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: International Conference on Software Engineering, pp. 364–374 (2009)
13. Perkins, J.H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W.F., Zibin, Y., Ernst, M.D., Rinard, M.: Automatically patching errors in deployed software. In: Symposium on Operating Systems Principles, pp. 87–102 (2009)
14. Georgas, J.C., van der Hoek, A., Taylor, R.N.: Architectural runtime configuration management in support of dependable self-adaptive software. In: Workshop on Architecting Dependable Systems, pp. 1–6 (2005)
15. Siegmund, N., Pukall, M., Soffner, M., Köppen, V., Saake, G.: Using software product lines for runtime interoperability. In: Workshop on AOP and Meta-Data for Software Evolution, pp. 1–7 (2009)
16. Ebnenasir, A.: Designing run-time fault-tolerance using dynamic updates. In: International Workshop on Software Engineering for Adaptive and Self-Managing Systems, p. 15 (2007)
17. 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)
18. Bondavalli, A., Chiaradonna, S., Cotroneo, D., Romano, L.: Effective fault treatment for improving the dependability of cots and legacy-based applications. IEEE Transactions on Dependable and Secure Computing 1(4), 223–237 (2004)
19. Clements, P., Northrup, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)
20. Qu, X., Cohen, M.B., Rothermel, G.: Configuration-aware regression testing: An empirical study of sampling and prioritization. In: International Symposium on Software Testing and Analysis, pp. 75–85 (July 2008)
21. Yilmaz, C., Cohen, M.B., Porter, A.: Covering arrays for efficient fault characterization in complex configuration spaces. IEEE Transactions on Software Engineering 31(1), 20–34 (2006)
22. Elkhodary, A., Esfahani, N., Malek, S.: FUSION: A framework for engineering self-tuning self-adaptive software systems. In: Proceedings of the International Symposium on the Foundations of Software Engineering (November 2010)
23. Kuhn, D., Wallace, D.R., Gallo, A.M.: Software fault interactions and implications for software testing. IEEE Transactions on Software Engineering 30(6), 418–421 (2004)
24. Munoz, F., Baudry, B.: Artificial table testing dynamically adaptive systems. Technical report, Institut National de Recherche en Informatique et en Automatique (2009)
25. Hassan, A., Holt, R.: The top ten list: Dynamic fault prediction. In: Proceedings of the 21st IEEE International Conference on Software Maintenance, ICSM 2005, pp. 263–272 (2005)

26. Kim, S., Zimmermann, T., Whitehead Jr, E., Zeller, A.: Predicting faults from cached history. In: Proceedings of the 29th International Conference on Software Engineering, pp. 489–498. IEEE Computer Society (2007)
27. Garvin, B.J., Cohen, M.B., Dwyer, M.B.: Using feature locality: can we leverage history to avoid failures during reconfiguration? In: Proceedings of the 8th Workshop on Assurances for Self-Adaptive Systems, ASAS 2011, pp. 24–33 (2011)
28. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 283–294 (2011)
29. Adler, M.: Mars exploration rover Spirit sol 18 anomaly. In: AIAA Space Conference/International Mars Conference (September 2004)
30. Batory, D.: Scaling step-wise refinement. IEEE Transactions on Software Engineering 30(6), 355–371 (2004)
31. Czarnecki, K., She, S., Wasowski, A.: Sample spaces and feature models: There and back again. In: International Software Product Line Conference, pp. 22–31 (2008)
32. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering. Springer, Berlin (2005)
33. Cohen, M.B., Dwyer, M.B., Shi, J.: Coverage and adequacy in software product line testing. In: Proceedings of the Workshop on the Role of Architecture for Testing and Analysis, pp. 53–63 (July 2006)
34. Cohen, M.B., Dwyer, M.B., Shi, J.: Interaction testing of highly-configurable systems in the presence of constraints. In: International Symposium on Software Testing and Analysis, pp. 129–139 (July 2007)
35. IEEE Standards Board: ANSI/IEEE Std 610.121990:Standard Glossary of Software Engineering Terminology. IEEE, New York (1990)
36. Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., Fox, A.: Microreboot — a technique for cheap recovery. In: OSDI 2004: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation, p. 3. USENIX Association, Berkeley (2004)
37. Strunk, E., Knight, J.: Assured reconfiguration of embedded real-time software. In: International Conference on Dependable Systems and Networks, pp. 367–376 (2004)
38. de Lemos, R., Fiadeiro, J.: An architectural support for self-adaptive software for treating faults. In: Proceedings of the First Workshop on Self-Healing Systems, WOSS 2002, pp. 39–42 (2002)
39. Garvin, B., Cohen, M.: Feature interaction faults revisited: An exploratory study. In: International Symposium on Software Reliability Engineering, pp. 90–99 (November 2011)
40. Free Software Foundation: GCC Bugzilla (March 2010), `http://gcc.gnu.org/bugzilla/`
41. Free Software Foundation: GNU 4.1.1 manpages (2005), `http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/`
42. International Organization for Standardization: ISO/IEC 9899: – Programming languages – C (September 2007)

Emerging Techniques for the Engineering of Self-Adaptive High-Integrity Software

Radu Calinescu

Department of Computer Science, University of York, UK
radu.calinescu@york.ac.uk

The journey of a thousand miles begins beneath one's feet.

Lao Tzu

Abstract. The demand for cost effectiveness and increased flexibility has driven the fast-paced adoption of software systems in areas where requirement violations may lead to financial loss or loss of life. Many of these software systems need to deliver not only high integrity but also self adaptation to the continual changes that characterise such application areas. A challenge long solved by control theory for continuous-behaviour systems was thus reopened in the realm of software systems. Software engineering needs to embark on a quest for *self-adaptive high-integrity software.* This paper explains the growing need for software capable of both self-adaptation *and* high integrity, and explores the starting point for the quest to make it a reality. We overview emerging techniques for the engineering of self-adaptive high-integrity software, propose a service-based architecture that aims to integrate these techniques, and discuss opportunities for future research.

1 Introduction

A growing number of software and software-controlled systems are built to adapt to changes in their environment, requirements and internal state. These *self-adaptive software systems* [19,56] can successfully reconfigure themselves in response to sensor-detected changes, typically through using a combination of heuristics, simulation and artificial intelligence techniques.

The development of successful self-adaptive software within hardly a decade since the advent of autonomic computing [35,38] is a remarkable achievement. Nevertheless, this achievement alone is insufficient for an important class of applications in which self-adaptive software plays an increasingly significant role. These are applications for which requirement violations may lead to loss of life or financial loss. Healthcare, transportation and finance are among the domains that rely on such *safety-critical* or *business-critical* applications.

Clearly, self-adaptive software used in safety-critical and business-critical applications must be characterised by *high integrity*—in the sense specified by the NIST definition [52]:

"*High integrity software is software that must be trusted to work dependably in some critical function, and whose failure to do so may have catastrophic results, such as serious injury, loss of life or property, business failure or breach of security.*"

This definition requires high-integrity software to "*work dependably*", which Meyer [48] equates with a combination of three properties:

1. correctness—compliance with the specification;
2. robustness—ability to withstand erroneous use outside the specification;
3. security—ability to withstand malicious use outside the specification.

As emphasised in a position paper [11] that motivated the work described here, software engineering tools for building software that is both correct and robust do exist. They include formal verification and validation (V&V), design by contract, and quality assurance [9,48,52].

However, for self-adaptive software, the three properties listed above must continue to hold as the software evolves to adapt to change. This additional requirement changes everything, because traditional software engineering approaches to developing high-integrity software were devised for off-line use during the design or V&V stages of the software lifecycle. As described in [11], "they operate with models, properties, assumptions and conjectures that in the case of self-adaptive software are unknown until the application is deployed and running—and which change over time." The range of changes that can affect self-adaptive software systems is extremely large or, in the case of large-scale complex systems like those discussed in [57], unbounded. Therefore, analysing the adaptation state space off-line is impractical in the first case, and unfeasible in the latter. Analogous techniques that can be applied automatically, while the system is running, are required for *self-adaptive high-integrity software.*

This challenge of simultaneous adaptation and high integrity has long been addressed by control theory, albeit primarily for continuous-behaviour systems (e.g., [26]). As cost savings and the need for increased flexibility have led to the replacement of these systems with software-based ones, the challenge is again open—for both software-only and embedded (or cyber-physical) systems.

The rest of the paper explores several aspects of self-adaptive high-integrity software, and discusses a service-based architecture for building software systems with these characteristics. Section 2 describes several archetypal applications that require self-adaptive high-integrity software. Section 3 overviews emerging software engineering techniques that support the development of self-adaptive high-integrity software, and discusses the current trend to implement critical software applications through the dynamic integration of heterogeneous services. Section 4 introduces a generic service-based software architecture that employs a combination of these techniques to produce *self-adaptive high-integrity software.* Finally, a preliminary research agenda is discussed in Section 5.

2 Critical Applications Requiring Self-Adaptive High-Integrity Software

This section overviews a selection of critical applications from three application domains, explaining why they each need software that supports both self adaptation and high integrity.

Healthcare. The fast ageing of the world's population is accommodated by many developed countries through healthcare budget increases that exceed the overall rate of economic growth. As this approach is unsustainable in the long term, IT-enabled *ambient assisted living* is perceived as an effective long-term solution for the monitoring of patients with chronic diseases and mobility-related conditions.

Ambient assisted living applications that employ wearable systems for health monitoring and use remote services for vital parameter analysis, medical record access, etc. could extend the time that elderly people manage independently at home, thus reducing healthcare costs and also improving their quality of life. Software-controlled systems integrating this 24-hour patient monitoring equipment with adaptive infusion pumps are envisaged as a potential extension of this solution [39,40]. (Infusion pumps are medical devices for the controlled delivery of medication and nutrients into a patient's body.) Medical conditions that could benefit from this approach include chronic cardiac and respiratory problems, diabetes, and high-risk pregnancies [33]. Nevertheless, software-controlled infusion pumps have a poor safety record even when used in a non-adaptive operating mode [58], so their integration into adaptive, closed-loop control solutions raises major concerns.

Transportation. In Europe alone, the transport sector is required to achieve "*a reduction of at least 60% of greenhouse gas emissions by 2050 with respect to 1990*" [22] as a contribution towards limiting climate change below 2°C. To achieve this objective, the manufaturers of next-generation vehicles and the planners of future road infrastructure will use safety–critical self-adaptive software to inform and help drivers respond to changes in traffic conditions, reducing travel time and fuel consumption, and improving road safety [30,34]. Despite significant advances in the underlying technology, security and reliability concerns have been raised about these applications [1,44].

Finance. In the finance industry, stock exchange transactions are increasingly carried out by automated trading systems that can react faster than their human counterparts. Furthermore, the adoption of adaptive, business-critical software trading agents in recent years has led to highly flexible applications whose effectiveness often matches that of human experts [21,37].

Nevertheless, self-adaptation in automated trading agents is a double-edged sword. Unsuitable adaptation might have been one of the causes of the still not fully explained 6th May 2010 Flash Crash that wiped $1 trillion in market value for a 20-minute period [24] and of the lower-impact but equally worrying 8.1% plunge in the natural gas price for 15 seconds on 8th June 2011 [49].

Table 1. Techniques that can help support the development and operation of self-adaptive high-integrity software

Technique/Research area	Description	Examples
models @ runtime	Models of the functional and/or non-functional software behaviour are analysed at runtime, in order to select system configurations that satisfy the requirements.	[8,29,31,50]
on-line learning	The parameters and/or structure of the models used to establish reliability, performance or functional properties of self-adaptive software are estimated at runtime, based on observations of the software behaviour.	[7,14,25,59]
quantitative model checking @ runtime	Non-functional software requirements are expressed as probabilistic temporal-logic properties, and are analysed at runtime, to predict or detect requirement violations and to guide adaptation.	[16,17,25,27,42]
runtime verification	Finite, partial execution traces are analysed formally to detect requirements violations, and the analysis may trigger runtime software adaptations.	[6,43,45,54]
runtime certification	The dependability of self-adaptive software is (re)certified after each runtime reconfiguration step.	[23,55]
model-driven development @ runtime	Runtime architectural changes are achieved through the on-line synthesis of the connectors required to include new software components into the adaptive system.	[7,10,20,36]

3 Background

3.1 Techniques for Self-Adaptive High-Integrity Software

This section (adapted from our previous work in [11]) describes the main research areas in which effort has been dedicated to the development of techniques that have the potential to support the realisation of self-adaptive high-integrity software. Table 1 summarises these results.

Models @ runtime. A growing number of research projects are investigating the use of models to steer the runtime adaptation of software systems. The types of models used by these projects range from architectural models [29,31] to parametric models of the valid system configurations [50] and data-flow automata [8].

The approach proposed in [29,31] employs formal analysis of architectural models in order to achieve software adaptation. In contrast, the "dynamic software product line" approach described in [50] achieves this runtime adaptation by starting with a collection of system configurations whose non-functional properties are analysed and quantified off-line. A technique called "aspect-oriented model reasoning' is then used at runtime, to select and adopt the optimal configuration according to a set of well-defined requirements. Finally, the approach in [8] uses synthesised data-flow automata to model the behaviour of web services and to support their automatic composition into software applications.

On-line learning. The effectiveness of model-based reasoning about the properties of a software system depends on the accuracy of the models used in the analysis. This dependency is particularly relevant for self-adaptive software, where the system evolution in response to changes can easily render obsolete the very models used to guide this evolution. This serious limitation is addressed by *on-line learning* techniques that use observations of the software behaviour to maintain the analysed models up to date.

The project presented in [7], for instance, is actively working on the development of a suite of statistical and automata learning techniques for inferring the functional semantics and the behavioural semantics of networked systems, respectively.

In the related approaches proposed in [14,25], the self-adaptation of service-based systems with strict reliability requirements is achieved through the analysis of discrete-time Markovian models whose transition probabilities are learnt on-line by using Bayesian learning techniques. An analogous method for predicting the response time of software components by using Kalman filter estimators is described in [59]. This method enables the use of accurate queueing models in the runtime analysis of the performance-related properties of certain types of self-adaptive software.

Quantitative model checking @ runtime. Recent research aimed at improving the dependability of self-adaptive software systems has proposed the use of quantitative model checking in the runtime adaptation process [12,13,16,17,25]. Quantitative model checking [41] is a mathematically-based technique for establishing the correctness, performance and reliability of systems characterised by stochastic behaviour.

Quantitative model checking is traditionally used for the off-line analysis of system properties expressed in temporal-logics extended with probabilities, costs and rewards. In the '@runtime' variant of the technique advocated in [13,16,17,25], this analysis is performed on-line, on continually updated versions of the software model and of its non-functional properties. The results of the analysis are used to guide adaptation in ways that guarantee that the software continues to satisfy its requirements despite changes in environment, workload and internal state. Maintaining the model up to date involves the application of the learning techniques described earlier in the paper [14,25,59], to ensure that model parameters (e.g., the transition probabilities of discrete-time Markov chains or the transition rates of continuous-time Markov chains) reflect the evolution of the software behaviour. In contrast, the updates in the analysed properties correspond to user-initiated modifications in the non-functional requirements of the software.

Given the potentially high overheads of quantitative model checking, using the technique successfully in a runtime setting requires the exploitation of recent research into improving its scalability [27,42]. The results presented in [27] achieve significant scalability improvements by precomputing the quantitative properties of the self-adaptive software off-line, as symbolic expressions whose parameters are the variable success and failure probabilities of the software components. The complementary approach in [42] works by restricting the runtime analysis to those parts of the model that are affected by change, and reusing the results from the previous analysis of all other parts.

Runtime verification. Runtime verification [45,47,54] is a technique that complements off-line testing with the runtime monitoring and extraction of finite software execution traces, followed by the analysis of these traces against a formal specification of the correct software behaviour. This specification is described using formalisms that range from temporal logics [45,54] and regular expressions [2] to state machines [4] and rule systems [5]. In extended variants of the technique, the runtime detection of violations in the software requirements is used to trigger adaptions that have a remedial effect [6,43].

Runtime verification is particularly suitable for self-adaptive software, where the ability to use off-line testing to identify requirement violations is even more limited than in the case of traditional, non-adaptive software.

Runtime certification. Runtime certification [55] refers to the on-line certification of the dependability of self-adaptive software. The technique aims to augment the fault detection, identification and reconfiguration approach from [23] with guarantees that the chosen software reconfigurations do not have a negative impact on dependability. The certification is achieved by means of model-based runtime verification.

Model-driven development @ runtime. Model-driven development @ runtime techniques were recently proposed [7,10,20,36] for the on-line synthesis of interfaces (or *connectors*) between the dynamically selected components of self-adaptive software systems. The approach is currently applicable to service-oriented software architectures, whose web service components expose standards-based WSDL "models" [7,10,20]. These models are used to synthesise the connectors required to integrate new components into an existing software architecture as part of the adaptation process, while the framework proposed in [36] enables the formal characterisation and verification of these connectors.

3.2 Critical Application Development through Service Integration

National and international strategic research agendas envisage that the types of safety-critical and business-critical applications described in Sections 1–2 will be increasingly developed through the dynamic integration of *software services* [28,51,53]. These services are expected to be flexible and shareable, to belong to multiple applications at the same time, and to self-adapt in response to change.

The research-funding programmes set up to support fundamental and applied research leading to the development of such services specify that they will need to interoperate across a range of platforms that includes private and public clouds, Internet of Things (IoT) and Internet of Contents (IoC). In other words, they need to be self-adaptive high-integrity services capable of on-the-fly integration into critical software applications that inherit the capabilities of their component services.

4 Towards a Service-Based Architecture Integrating "@ Runtime" Techniques for Self Adaptation and High Integrity

The vision of service-based future critical applications described above is illustrated in Figure 1, which depicts the high-level architecture of two of the applications mentioned in Section 2. The first application is an ambient assisted living system assembled from:

– wearable vital parameter monitoring (e.g., [3]) and infusion pump (e.g., [46]) IoT devices;
– medical record (MR) and vital parameter analysis (VPA) services running on a private cloud;
– public-cloud services such as emergency (Emerg), pharmacy (Pharm), accident and emergency (A&E), weather forecast (WF) and roadmap (Rmap).

The last two of these services are also part of a road traffic information system, which also comprises:

– smart-vehicle and traffic-sensor IoT components;
– private-cloud traffic analysis (TA) services.

Each legacy or newly developed component of the service-based software architecture from Figure 1 is wrapped into an appropriately configured instance of a reusable *self-adaptive high-integrity service*. This should be a standards-based service that augments traditional service-oriented architecture (SOA) functionality with enhanced versions of the techniques described in Section 3.1, therefore enabling the transparent integration of these "@runtime" techniques into critical software applications.

Figure 2 shows a possible prototype architecture for such a self-adaptive high-integrity service. The elements of this architecture employ the "@runtime" techniques from Section 3.1 as described below.

Self-adaptive high-integrity middleware. The application-independent, reconfigurable self-adaptive high-integrity middleware at the core of the architecture continually learns, maintains and exploits detailed, accurate and up-to-date behavioural models of peer services and of the system components it provides a wrapper for. To achieve this, the middleware uses a combination of:

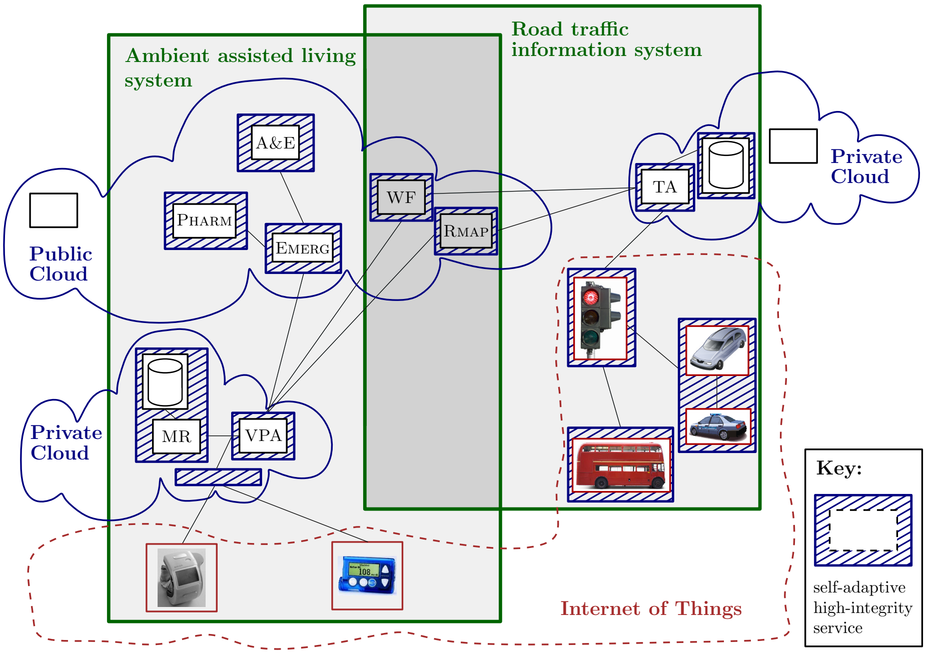

Road traffic information system
Ambient assisted living system
Private Cloud
A&E
WF
TA
PHARM
RMAP
EMERG
Public Cloud
Private Cloud
MR
VPA
108
Key:
self-adaptive high-integrity service
Internet of Things

Fig. 1. Critical applications assembled through the cross-platform integration of interoperable services

(a) "models @ runtime" (to maintain a set of models that reflect the evolution of its own behaviour and of the behaviour of its peer services);
(b) on-line learning techniques (to update its models);
(c) quantitative model checking @ runtime, and runtime verification techniques (to guide its dependable adaptation); and
(d) runtime certification (to certify itself for the benefit of peer services and of the applications it belongs to).

Configurator. The domain-specific configurator is used to repurpose the self-adaptive high-integrity middleware for the application domain (or domains) that it is meant to operate in. Configured middleware will be able to "speak" the relevant domain-specific language(s) with similarly configured peer services, and with the administrators and users of applications from these domains. This will enable, for instance, the exchange and automated translation of domain-specific requirements into an internal representation that can be analysed automatically against up-to-date, on-line learnt models. The use of model-driven development @ runtime techniques will be required to synthesise the software modules supporting this functionality.

Intelligent proxies. The intelligent proxies linking the services belonging to the same critical application(s) represent significantly enhanced versions of the

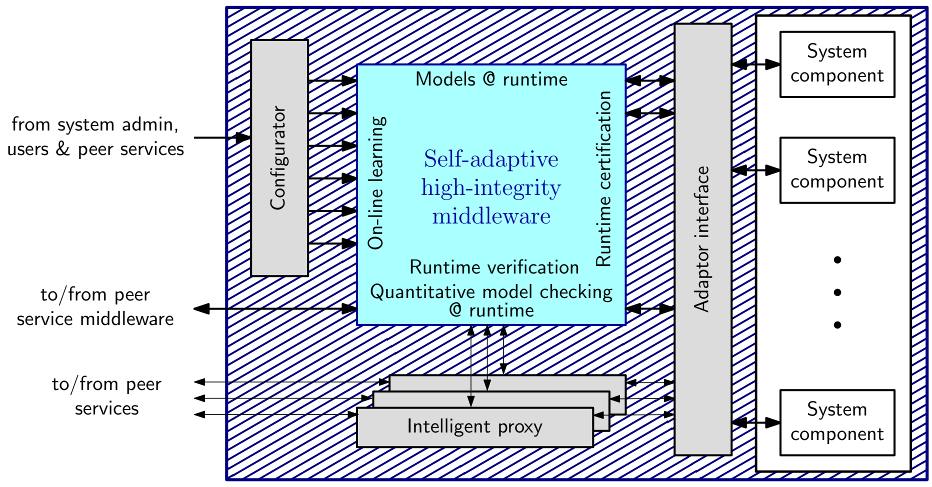


Fig. 2. Prototype architecture of a self-adaptive high-integrity service that uses the "@runtime" techniques from Section 3.1

traditional web service proxies used in today's SOA applications. Thus, in addition to ensuring service interoperability, the intelligent proxies will continually monitor the performance and dependability properties (e.g., response time and success rate) of peer services. They will use the data obtained from this monitoring and fast on-line learning algorithms to devise partial peer-service models that will be assembled into fully-fledged behavioural models by the self-adaptive, high-integrity middleware.

Adaptor interfaces. The application-specific adaptor interfaces connect heterogeneous system components (e.g., the IoT virtual objects and cloud-deployed services from the applications in Figure 1) to the middleware modules. As a result, such components:

(a) can benefit from the capabilities provided by the middleware; and
(b) can be organised into interoperable self-adaptive, high-integrity services for integration into multi-platform critical applications.

5 Conclusion

We argued that software engineering is unprepared for today's fast-paced adoption of self-adaptive software in safety-critical and business-critical applications. The existing approaches to engineering the high-integrity software required in such applications rely on models and properties that do not change over time, and are underpinned by high-overhead analysis techniques suited for off-line use. Neither of these prerequisites holds for self-adaptive software, which is typically developed using "best-effort" techniques that cannot guarantee requirements compliance.

Software engineering needs to embark on a quest for techniques capable of delivering high integrity and self adaptation at the same time. The outcome of this quest should include low-overhead software engineering techniques capable of fully automated, on-line operation, and novel architectures that integrate these techniques into high-integrity self-adaptive software systems.

We explored a number of emerging software engineering paradigms that collectively represent the starting point for this quest. The core principle underlying all these paradigms is that software engineering techniques traditionally used in the off-line stages of the software life cycle should be complemented by analogous techniques that are suitable for use at run time. A growing number of research projects work on new software engineering techniques that match this description. They include projects that use *models at runtime* to support the dependable evolution, formal analysis, and certification of self-adaptive software; and projects concerned with learning and updating the parameters and structure of these models continually.

Many challenges need to be overcome before we can achieve effective assurances for critical applications that use self-adaptive software. Key among these challenges is the need for runtime model analysis and verification techniques that are lightweight, incremental and compositional [15,18,32]. The ability to take full advantage of such techniques will depend on the successful development of effective approaches for the on-line learning of the analysed models.

Future software systems will comprise components that join and leave dynamically [7,57], so suitable software components will need to be discovered and their behaviour will need to be learnt "on the fly". Assembling these software systems for use in critical applications will require software architectures based on reconfigurable middleware that integrates state-of-the-art runtime learning and analysis techniques into an easy-to-use framework. We suggested a possible service-based architecture for this role and indicated how it could be built through integrating a number of emerging software engineering techniques, but significant additional work is required in this area.

Another important challenge is the development of novel approaches for achieving the levels of component interoperability required by high-integrity self-adaptive software. These approaches will have to be based on new standards for expressing a broad range of functional and non-functional properties of software components, and on scalable techniques for inferring the system-level properties from the component properties.

Future safety-critical and business-critical applications will comprise large numbers of embedded (or *cyber-physical*) systems. Ensuring that these applications achieve both high integrity and self adaptation will require the integration of the software engineering advances mentioned above with established control theory techniques.

Last but not least, there is the problem of "who watches the watchmen": the intelligent future middleware underpinning the high-integrity self-adaptive software systems of the future will itself need to be certified or self-certifiable.

Acknowledgements. This work was partly supported by the UK Engineering and Physical Sciences Research Council grant EP/H042644/1. The author is grateful to Octavian Pastravanu for the insightful discussion about control theory approaches to developing continuous-behaviour self-adaptive systems.

References

1. Aijaz, A., Bochow, B., Dotzer, F., Festag, A., Gerlach, M., Kroh, R., Leinmuller, T.: Attacks on inter vehicle communication systems - an analysis. In: Proc. 3nd Intl. Workshop Intelligent Transportation, pp. 189–194 (2006)
2. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2005), pp. 345–364. ACM (2005)
3. Anliker, U., et al.: AMON: a wearable multiparameter medical monitoring and alert system. IEEE Transactions on Information Technology in Biomedicine 8(4), 415–427 (2004)
4. Barringer, H., Havelund, K.: TraceContract: A Scala DSL for Trace Analysis. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 57–72. Springer, Heidelberg (2011)
5. Barringer, H., Havelund, K., Rydeheard, D., Groce, A.: Rule Systems for Runtime Verification: A Short Tutorial. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 1–24. Springer, Heidelberg (2009)
6. Bauer, A., Leucker, M., Schallhart, C.: Model-based methods for the runtime analysis of reactive distributed systems. In: Proc. Australian Software Engineering Conference, pp. 243–252 (2006)
7. Bennaceur, A., Howar, F., Issarny, V., Johansson, R., Moschitti, A., Spalazzese, R., Steffen, B., Sykes, D.: Machine Learning for Emergent Middleware. In: Proceedings of the Joint Workshop on Intelligent Methods for Software System Engineering (2012)
8. Bertolino, A., Inverardi, P., Pelliccione, P., Tivoli, M.: Automatic synthesis of behavior protocols for composable web-services. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pp. 141–150. ACM (2009)
9. Burton, S., Clark, J., Galloway, A., McDermid, J.: Automated V&V for high integrity systems, a targeted formal methods approach. In: NASA Langley Formal Methods Workshop (January 2000), `ftp://ftp.cs.york.ac.uk/pub/hise/NASALangley.pdf` (last retrieved on September 10, 2012)
10. Calinescu, R.: Run-time connector synthesis for autonomic systems of systems. Journal On Advances in Intelligent Systems 2(2-3), 376–386 (2009)
11. Calinescu, R.: When the requirements for adaptation and high integrity meet. In: Proceedings of the 8th Workshop on Assurances for Self-Adaptive Systems (ASAS 2011), pp. 1–4. ACM, New York (2011)
12. Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G.: Dynamic QoS management and optimisation in service-based systems. IEEE Transactions on Software Engineering 37(3), 387–409 (2011)

13. Calinescu, R., Ghezzi, C., Kwiatkowska, M., Mirandola, R.: Self-adaptive software needs quantitative verification at runtime. Communications of the ACM 55(9), 69–77 (2012)
14. Calinescu, R., Johnson, K., Rafiq, Y.: Using observation ageing to improve Markovian model learning in QoS engineering. In: Proceedings 2nd ACM/SPEC International Conference on Performance Engineering, pp. 505–510 (2011)
15. Calinescu, R., Kikuchi, S.: Formal Methods @ Runtime. In: Calinescu, R., Jackson, E. (eds.) Monterey Workshop 2010. LNCS, vol. 6662, pp. 122–135. Springer, Heidelberg (2011)
16. Calinescu, R., Kwiatkowska, M.: CADS*: Computer-Aided Development of Self-* Systems. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 421–424. Springer, Heidelberg (2009)
17. Calinescu, R., Kwiatkowska, M.: Using quantitative analysis to implement autonomic IT systems. In: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), pp. 100–110 (2009)
18. Calinescu, R., Kikuchi, S., Johnson, K.: Using Compositional Verification to Manage Change in Large-Scale Complex IT Systems. In: Large-Scale Complex IT Systems - Development, Operation and Management. LNCS, vol. 7539, pp. 303–329. Springer (2012)
19. 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)
20. Cavallaro, L., Di Nitto, E., Pelliccione, P., Pradella, M., Tivoli, M.: Synthesizing adapters for conversational web-services from their WSDL interface. In: ICSE 2010 SEAMS: Workshop on Software Engineering for Adaptive and Self-Managing Systems, pp. 104–113 (2010)
21. Collins, J., Ketter, W., Gini, M.: Flexible decision control in an autonomous trading agent. Electronic Commerce Research & Appl. 8(2), 91–105 (2009)
22. COM(2011) 144: European Commission. Roadmap to a Single European Transport Area Towards a competitive and resource efficient transport system (2011), `http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=COM:2011:0144:FIN:EN:PDF` (last retrieved on September 10, 2012)
23. Crow, J., Rushby, J.: Model-based reconfiguration: Diagnosis and recovery. NASA Contractor Report 4596, NASA Langley Research Center, Hampton, VA (Work performed by SRI International) (May 1994)
24. Easley, D., de Prado, M.M.L., O'Hara, M.: The microstructure of the 'Flash Crash': Flow toxicity, liquidity crashes and the probability of informed trading. Journal of Portofolio Management 37(2), 118–128 (2011)
25. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by runtime adaptation. In: Proceedings of the 31st International Conference on Software Engineering, pp. 111–121. IEEE Computer Society Press (2009)
26. Feng, G., Lozano, R.: Adaptive Control Systems. Elsevier (1999)
27. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: Proceedings of the 33rd International Conference on Software Engineering, IEEE Computer Society (2011)

28. Future Internet Assembly. Research Roadmap Towards Framework 8: Research Priorities for the Future Internet (2011), `http://fisa.future-internet.eu/images/0/0c/Future_Internet_Assembly_Research_Roadmap_V1.pdf`
29. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using architecture models for runtime adaptability. IEEE Software 23, 62–70 (2006)
30. Fritsch, S., Senart, A., Schmidt, D.C., Clarke, S.: Time-bounded adaptation for automotive system software. In: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 571–580. ACM, New York (2008)
31. Garlan, D., Schmerl, B.R.: Using architectural models at runtime: Research challenges. In: European Workshop Software Architecture, pp. 200–205 (2004)
32. Ghezzi, C.: Evolution, adaptation and the quest for incrementality. In: Preproceedings of the 17th Monterey Workshop on Development, Operation and Management of Large-Scale Complex IT Systems, pp. 79–88 (2012)
33. Ghini, V., Ferretti, S., Panzieri, F.: M-Hippocrates: Enabling Reliable and Interactive Mobile Health Services. IT Professional 14(3), 29–35 (2012)
34. Hartenstein, H., Laberteaux, K.P. (eds.): VANET: Vehicular Applications and Inter-Networking Technologies. John Wiley & Sons (2009)
35. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing—degrees, models, and applications. ACM Comp. Surveys 40(3), 1–28 (2008)
36. Issarny, V., Bennaceur, A., Bromberg, Y.-D.: Middleware-Layer Connector Synthesis: Beyond State of the Art in Middleware Interoperability. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 217–255. Springer, Heidelberg (2011)
37. Izumi, K., Toriumi, F., Matsui, H.: Evaluation of automated-trading strategies using an artificial market. Neurocomputing 72(16-18), 3469–3476 (2009)
38. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer Journal 36(1), 41–50 (2003)
39. Kovatchev, B.: Closed loop control for type 1 diabetes. British Medical Journal 342, d1911 (2011)
40. Kramer, G.C., Kinsky, M.P., Prough, D.S., Salinas, J., Sondeen, J.L., Hazel-Scerbo, M.L., Mitchell, C.E.: Closed-loop control of fluid therapy for treatment of hypovolemia. Journal of Trauma-Injury Infection & Critical Care 64(4), S333–S341 (2008)
41. Kwiatkowska, M.: Quantitative verification: Models, techniques and tools. In: Proc. 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. Foundations of Software Engineering, pp. 449–458. ACM Press (2007)
42. Kwiatkowska, M., Parker, D., Qu, H.: Incremental quantitative verification for Markov decision processes. In: Proceedings 2011 IEEE/IFIP International Conference Dependable Systems and Networks (2011)
43. Kyas, M., Prisacariu, C., Schneider, G.: Run-Time Monitoring of Electronic Contracts. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 397–407. Springer, Heidelberg (2008)
44. Lee, U., Cheung, R., Gerla, M.: Emerging vehicular applications. In: Olariu, S., Weigle, M.C. (eds.) Vehicular Networks: From Theory to Practice. Chapman and Hall/CRC (2009)
45. Leucker, M., Schallhart, C.: A brief account of runtime verification. Journal of Logic and Algebraic Programming 78(5), 293–303 (2009)

46. Mastrototaro, J., Lee, S.: The Integrated MiniMed Paradigm Real-Time Insulin Pump and Glucose Monitoring System: Implications for Improved Patient Outcomes. Diabetes Technology & Therapeutics 11(s1), 37–43 (2009)
47. Meredith, P., Roşu, G.: Runtime Verification with the RV System. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 136–152. Springer, Heidelberg (2010)
48. Meyer, B.: Dependable Software. In: Kohlas, J., Meyer, B., Schiper, A. (eds.) Dependable Systems: Software, Computing, Networks. LNCS, vol. 4028, pp. 1–33. Springer, Heidelberg (2006)
49. Meyer, G.: Traders flummoxed by natural gas 'flash crash'. Financial Times (June 9, 2011)
50. Morin, B., Barais, O., Jezequel, J.-M., Fleurey, F., Solberg, A.: Models@run.time to support dynamic adaptation. Computer 42, 44–51 (2009)
51. Networked European Software and Services Initiative. Research Priorities for the next Framework Programme for Research and Technological Development FP8 (May 2011), `http://www.nessi-europe.com/files/Docs/NESSI%20SRA_update_May_2011_V1-0.pdf`
52. Wallace, D.R., Ippolito, L.M., Kuhn, D.R.: High Integrity Software Standards and Guidelines. NIST SP 500-204, National Institute of Standards and Technology, Gaithersburg, MD, 20899 (September 1992)
53. National Science Foundation. Cyberinfrastructure Framework for 21st Century Science and Engineering. A Vision and Strategy for Data in Science, Engineering, and Education (April 2012), `http://www.nsf.gov/od/oci/cif21/DataVision2012.pdf`
54. Pnueli, A., Zaks, A.: PSL Model Checking and Run-Time Verification Via Testers. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 573–586. Springer, Heidelberg (2006)
55. Rushby, J.: Runtime Certification. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 21–35. Springer, Heidelberg (2008)
56. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2), 1–42 (2009)
57. Sommerville, I., Cliff, D., Calinescu, R., Keen, J., Kelly, J.T., Kwiatkowska, M., McDermid, J., Paige, R.: Large-scale complex IT systems. Communications of the ACM 55(7), 71–77 (2012)
58. Food, U.S.: Drug Administration — Center for Devices and Radiological Health. Infusion pump improvement initiative, White paper (April 2010), `http://www.fda.gov/MedicalDevices/ProductsandMedicalProcedures/GeneralHospitalDevicesandSupplies/InfusionPumps/ucm205424.htm` (last retrieved on September 10, 2012)
59. Zheng, T., Woodside, M., Litoiu, M.: Performance model estimation and tracking using optimal filters. IEEE Transactions on Software Engineering 34(3), 391–406 (2008)

Assurance of Self-adaptive Controllers for the Cloud

Alessio Gambi[1], Giovanni Toffetti[1], and Mauro Pezzè[1,2]

[1] University of Lugano 6904, Lugano, Switzerland
[2] University of Milano Bicocca 20100, Milan, Italy
{alessio.gambi,toffettg,mauro.pezze}@usi.ch

Abstract. In this paper we discuss the assurance of self-adaptive controllers for the Cloud, and we propose a taxonomy of controllers based on the supported assurance level. Self-adaptive systems for the Cloud are commonly built by means of controllers that aim to guarantee the required quality of service while containing costs, through a careful allocation of resources. Controllers determine the allocation of resources at runtime, based on the inputs and the status of the system, and referring to some knowledge, usually represented as adaptation rules or models. Assuring the reliability of self-adaptive controllers account to assuring that the adaptation rules or models represent well the system evolution. In this paper, we identify different categories of control models based on the assurance approaches. We introduce two main dimensions that characterize control models. The dimensions refer to the flexibility and scope of the system adaptability, and to the accuracy of the assurance results. We group control models in three main classes that depend on the kind of supported assurance that may be checked either at design or runtime. Controllers that support assurance of the control models at design time privilege reliability over adaptability. They usually represent the system at a high granularity level and come with high costs. Controllers that support assurance of the control models at runtime privilege adaptability over reliability. They represent the system at a finer granularity level and come with reduced costs. Controllers that combine different models may balance verification at design and runtime and find a good trade off between reliability, adaptability, granularity and costs.

1 Introduction

The Cloud paradigm allows for a more efficient use of computing resources, by decoupling software applications from their execution environment. The Cloud infrastructure disconnects applications from the execution environments by introducing a stack of abstraction layers that isolate the execution infrastructure –Infrastructure as a Service (IaaS)– the overall platform –Platform as a Service (PaaS)– and the provided services –Software as a Service (SaaS)–[1].

In this chapter, we focus on the IaaS layer that takes care of allocating resources to applications. Applications shall guarantee the qualities specified by their service level agreements (SLA), which usually associate penalties to SLA violations. At the same time resources come with costs, and providers aim to minimize costs, in order to increase competitiveness and profit [19]. To cope with an unpredictable set of combinations of application requests, service level agreements and usage patterns, the IaaS layers implement self-adaptive controllers, that is, controllers that adapt to different scenarios of applications to be executed, service requirements and traffic conditions.

1.1 An Example of Dynamic Resource Provisioning in the Cloud

To decide how to efficiently allocate resources, self-adaptive controllers refer to some knowledge that is provided in the form of either rules or models. Self-adaptive controllers use rules or models to evaluate different strategies, and chose the best possible tactic to cope with a degrading quality of service in the presence of varying traffic conditions, while avoiding indirect interferences between applications. Dually, self-adaptive controllers use rules or models to chose the right strategy to cope with increasing costs due to overallocated resources when traffic and application conditions change.

We exemplify the problem of devising controllers for dynamic resource allocation in the Cloud, by reporting a brief experience in managing an elastic application based on the *Sun Grid Engine (SGE)* middleware. SGE follows a standard Grid computing architecture with a singleton master node and a set of executor nodes: The master receives jobs that are dispatched to the executors that run them. The middleware has been deployed in a Cloud infrastructure, where virtual execution nodes can be allocated dynamically.

In Figure 1, we report the results of different runs of the system subject to an identical workload but different controllers. The workload lasts thirty minutes and fluctuates in time. It consists of video conversion jobs that execute in six seconds on average. The SLA of the application consists of a single SLO over the response time, and specifies that the system must complete the jobs in less than two minutes in average. The goal of the controller is to dynamically adjust the number of executor VMs to respect the SLO while minimizing costs that depend on the number of used VMs.

The plot depicts the workload in terms of requests per period, number of active executors, number of jobs in the system, and measured response time for two different runs. The continuous red line represents the system when controlled by a self-adaptive Kriging-based controller as described in [27]. We can see that the number of executors changes over time and the response time is always below the 2-minute threshold, marked with the dotted blue line in the figure.

The dashed green line represents the behaviour of a state-of-the-art static rule-based controller as presented by Rodero-Merino et al. in [22]. In the experiment, we set the scaling up rule threshold for the queue length to 15 jobs per executor: each time the ratio between the number of jobs in the queue divided by the number of active executors exceeds this threshold, the system spawns a new executor VM instance. We set the scaling down threshold to 5 jobs per executor. We can see that SLA is violated in correspondence of the second peak in the

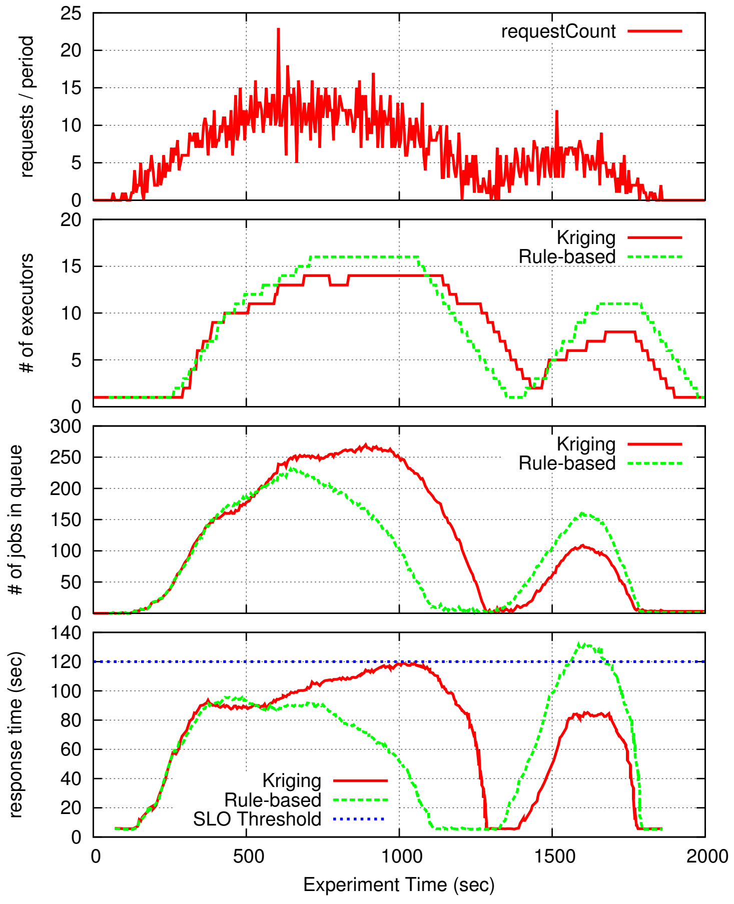


Fig. 1. Configuration, queue, and response time evolution for the SGE using Kriging-based and rule-based control

workload. This suggests that, albeit the rule-based system seems to scale up quickly enough in correspondence of the first workload peak, although using more resources in comparison to the Kriging-based controller, it does not cope well wit the second peak.

The example gives an intuitive idea of the type of actions and decisions that a cloud controller is required to take. The example is deliberately simple, since it has only one controlled variable, i.e., the number of executor nodes, and considers a single type of request. Typical Cloud-based applications are much more complex, since they combine different types of components and services, and hence have a large configuration space. Moreover, the range of served requests typically involves different sets of components causing possible software or hardware contentions that eventually impact on the performance of the applications. To this end, the application *workload mix*, that is the number and type of incoming requests, typically has a considerable effect on the application response time.

The assurance of self-adaptive controllers requires examining a potentially unlimited set of unpredictable configurations that arise when the system adapts to different execution conditions. While designing self-adaptive controllers is a hard and challenging job, the assurance of self-adaptive controllers is an even harder task, since assurance techniques must cope with infinitely many unpredictable configurations.

In this chapter, we identify two main dimensions of this problem, the target levels of assurance and adaptability, and we propose a classification of self adaptive controllers induced by these two dimensions. We argue that rules and models defined at design time privilege assurance over adaptability, being statically verifiable, but incapable of dealing with situations not foreseen at design time. On the other hand, rules and models that can change at runtime privilege adaptability over assurance, being able to deal with unpredictable situations, but verifiable only under certain conditions. We identify combinations of design and runtime elements that reach a good compromise between assurance and adaptability, and we distinguish some outliers that come from particular choices or uses.

This chapter is organized as follows. Section 2 discusses the many dimensions of the problem and introduces an assurance-base taxonomy for self-adaptive controllers. Section 3 overviews the main approaches based on rules or models defined at design time. Section 4 presents the main approaches where models and rules are adapted at runtime. Section 5 discusses combinations of different kinds of approaches. Section 6 indicates the main research directions in the field.

2 Assurance and Adaptability

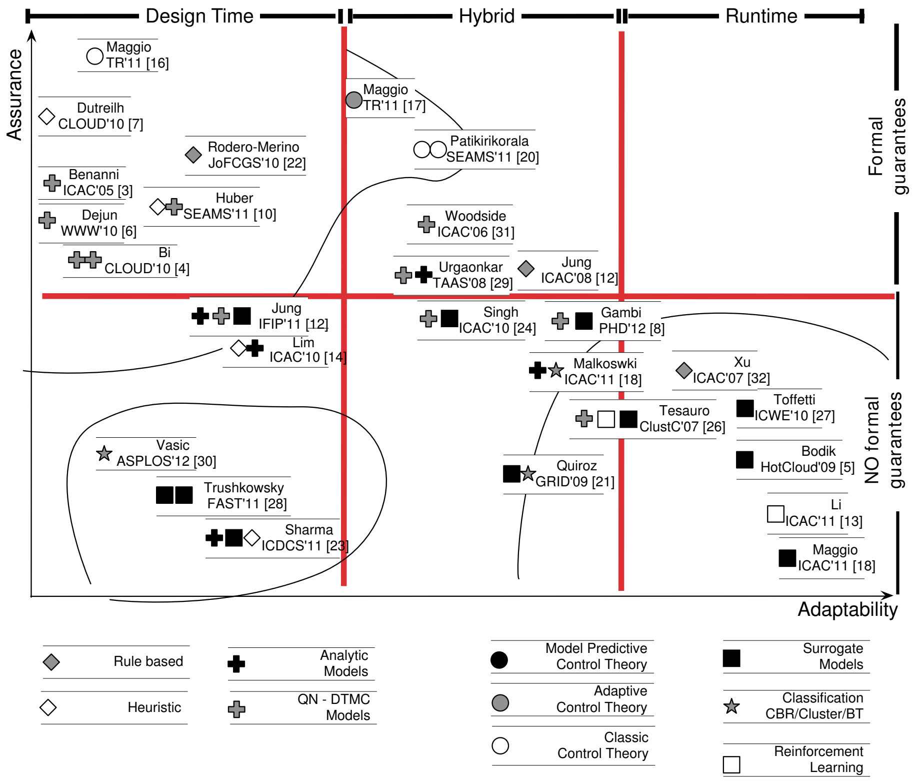
When assigning resources to applications, the Cloud shall solve the dilemma of allocating as many resources as possible, to reduce the violations of the service level agreement, while allocating as few resources as possible, to increase the profit and optimize the resource usage. The variety of available resources with different characteristics and costs, the variability and unpredictability of workload conditions, and the different effects of various configurations of resource allocations make the problem extremely hard if not impossible to solve algorithmically at design time. Self-adaptive controllers aim to identify suitable allocations of resources at runtime, based on some knowledge encapsulated in the controllers in the form of rules or models.

The problem of building efficient self-adaptive controllers has been the target of several research projects, which resulted in many approaches that differ for the models used to capture the knowledge of the system and for the strategies to identify a suitable configuration while reacting to changes of workload conditions. The models of system behavior that are used in self-adaptive controllers span from simple rules to complex analytic or surrogate models. Some approaches require models to be defined and tuned at design time, others define and tune models at runtime, depending on the nature of the models, the complexity of the system and the reliability requirements of the controllers. Rules and analytic models, like queuing networks, must be defined and tuned at design time and hardly adapt to unpredictable configurations, while surrogate models, like Kriging models, may be defined and tuned at runtime and can adapt to emerging scenarios. Defining proper analytic models for complex systems may require a considerable effort, but their reliability can be assured analytically or experimentally before deployment. Surrogate models can be tuned dynamically according to the system behavior either at testing or runtime independently from the complexity of the modeled system, but their reliability can be assured only on the basis of some hypothesis on the behavior of the system that may not be statically verifiable.

In this paper we argue that there is no best model for self-adaptive controllers, rather different models can be used for different purposes and under different conditions. We followed a systematic process to identify a set of representative approaches in literature that we use here as a reference. The first step of the process consisted of surveying the papers published in the main journals (ACM and IEEE Transactions) and conferences (ICAC, ICSE, Cloud, HotCloud, ACDC, GRID, ICDCS, ASPLOS) relevant for Cloud-computing and distributed or adaptive systems, and identifying the main proposals to deal with scaling or resource-provisioning controllers. Among these approaches, we identified a set of common dimensions (for example, the type of models used, the control logic, the artificial intelligence technique) that we used for clustering the different work. For each cluster we identified a representative by choosing the approach that is more directly applicable to cloud-computing controllers and that is supported by experimental results, and we use the representative to indicate the set of proposals.

We propose a taxonomy of controllers built along two main axes: assurance and adaptability. *Assurance* refers to the possibility of measuring the predictability of the runtime behavior, and thus to the level of reliability of the system. *Adaptability* refers to the capability of the model to cope with changing and evolving configurations, and thus to the level of flexibility and learning capabilities of the system. We chose these two criteria since they provide a clear representation of the main trade-off one has to face when choosing a controller approach. On one hand it is desirable for a controller to be adaptable, to complement the information available at design time with monitoring data from the running system, to adapt to the newly gathered information, and to continuously learn the most appropriate control behavior. On the other hand, a continuously adapted control behavior might practically diverge from the control desired at design time violating some important assumptions or requirements and providing no formal guarantees on its properties.

Assurance. We estimated the *assurance* level of the different approaches by considering the formality of the control model, the hypothesis on the system behavior and the correspondence between the system and the model:



The surveyed approaches represented in the space defined by *adaptability* and *assurance* dimensions. Different symbols represent different categories of approaches.

Fig. 2. A taxonomy of approaches for self-adaptive controllers

Formality of the Control Model. This accounts for the amount of formal guarantees that the approach gives on the model or control. For instance, in traditional control theory, under proper assumptions, the controlled system can be guaranteed to converge to an equilibrium point within a given range of oscillations.

Hypothesis on the System Behavior. This accounts for the set of assumptions on the system behavior that need to be valid for the approach to correctly model the system. For instance, control theory assumes system linearity, while Kriging models assume a certain degree of smoothness of the modeled system.

Correspondence between the System and the Model. This accounts for the (timely) coherence between the current state of the system and its runtime models.

Adaptability. We estimated the *adaptability* of the different approaches by considering three main aspects:

Support for Adaptation. This accounts for the amount of details of the model that are fixed at *design-time*, the amount of details that can be partially adapted or extended by combining additional models (*hybrid*), and the amount of detailed that the controller can learn from system monitoring data collected at *runtime.*

Degree of Adaptation. This accounts for the number of parameters and the degrees of freedom that can be changed in a model, spanning from fixed models, to linear-, polynomial-, regression models all the way to non-parametric models (for example, splines).

Frequency of Adaptation. This accounts for the costs of retraining in terms of speed and frequency, and measures how well a controller can use new monitoring information.

Figure 2 illustrates the two-dimensional space induced by the metrics we chose for classification. The adaptability axis is partitioned in three sections that correspond to approaches that are implemented at design time and have no provision for being updated, hybrid approaches that typically combine several models with limited adaptation capability, and solutions supporting runtime update of the control logic / models, respectively. Among these partitions, controllers with higher degrees of adaptation, or higher frequency of adaptation are moved to the right. The assurance axis is partitioned in two sections corresponding to the approaches that provide formal guarantees on the controller behavior, and the one that do not come with formal proof mechanisms.

Figure 2 presents a dispersion graph of the approaches organized along the two dimensions. In the figure, the approaches are indicated by means of the first author of the paper that bet represents the approach, the publication venue, and the reference number. We specify the nature of the proposed approaches by means of symbols that are explained in the labels.

The regions roughly identified in the figure emphasize easily identifiable similarities in the considered approaches which can be coarsely classified as being either analytic, black box, rule based, or hybrid.

Analytic. Analytic approaches tend to privilege reliability over adaptability. They rely on models that can be statically analyzed to formally prove stability and other important properties of the controllers, and thus they usually provide a high assurance level. However, they must be defined at design time, typically come with high costs in terms of modeling and system knowledge, and are only applicable under strong assumptions (for example, at least 'local' system linearity in control theory), and thus come with reduced adaptability. These models tend to accumulate in the top left corner of the figure.

Black Box. Approaches that use black box models tend to privilege adaptability over reliability. They rely on surrogate models built from monitoring data that can be automatically updated at runtime, while the system is running. Thus, they can adapt to situations that arise only at runtime resulting in high adaptability. However, they provide little support for analysis and formal assurance, and rely mostly on assumptions about the system behavior that may not be always easy to check statically and enforce dynamically. Thus they provide a lower assurance level than analytic approaches. In the figure these models occupy two distinct regions, the bottom right and the bottom left areas. The bottom right region corresponds to approaches based on black box models initialized and updated at runtime that can achieve high adaptability albeit at the price of a lesser assurance level. The bottom left region corresponds to approaches based on black box models that are used in a framework where they are initialized and updated at design time and not at runtime, thus giving up the adaptability that derive from the possibility of adapting to emerging scenarios at runtime, without gaining much in terms of assurance level.

Rule Based. Approaches based on rules appear in different flavors and span all over the taxonomy space depending on whether rules are fixed at design time and do not change, or new rules are discovered and possibly applied after approval, or the rule set is updated at runtime.

Hybrid. Approaches that combine design time and runtime techniques aim to conjugate the adaptability of models used at runtime with the assurance gained with models used at design time. In the figure, they occupy the middle region.

In the following sections, we illustrate the different classes of the approaches considering the adaptability axis. Section 3 discusses static (design-time) approaches, Section 4 presents approaches that support runtime updates, and Section 5 overviews hybrid approaches that combine both solutions. Each section is organized by clustering approaches that have some commonality (for example, in the type of control logic, in the nature of the models). For each cluster we outline a relevant set of sample approaches, and discuss in detail few representative ones, stressing the relations and trade-offs between assurance level and adaptability.

3 Static Approaches and Design Time Assurance

We use the term *static approaches* to indicate approaches based on models of the controlled system that are defined and verified at design time and not modified, updated, or tuned in production, after system deployment. We refer to these models as static models to stress the design nature of their construction and analysis. These models privilege design time assurance over runtime adaptability: The models of the system identified at design time are not changed during runtime activities to preserve the validity of the design time proofs used to identify system *viability zones* that are defined as states in which the system operation is not compromised [25], along with the main properties of the system behavior.

The most popular static approaches are based on either analytical models or rules and thresholds, while only some static approaches are based based on black-box and surrogate models defined and tuned before deployment. Analytical models can be divided in (1) *time invariant* models that describe the system steady state behavior, and (2) *time varying* models that describe the evolution and the transitory phases of the system behavior. Time invariant models are mathematical relations derived by analyzing queuing networks, Petri nets and Markov chains, time varying models are typically dynamic state-equations commonly used in classic control theory.

Rules and threshold are constructs building upon logical expressions that are used to trigger control actions. Controllers implemented by means of rules and thresholds are amenable to being transformed into formal representations and proven to satisfy, at least to some extent, predictable behavioral properties. For example, one can prove the absence of conflicts and contradictions in sets of rules, and the termination of the rule triggering process.

Black-box and surrogate models are derived from empirical data obtained by executing the systems. They capture the relationships between input and output variables and are commonly used by controllers as oracles, that is, to predict possible system behaviors in terms of the same input/output variables. Although black-box and surrogate models are amenable to runtime adaptation, in some cases they are used as the core of static approaches. These approaches do not consider the additional adaptability that derive from these models, rather, they focus on the cost of deriving accurate analytical models that decreases dramatically when referring to black-box and analytic models.

Being defined at design time and not updated at runtime, static approaches rely heavily on the correctness and completeness of the knowledge encoded in their models. This implies an extensive degree of experience and understanding of the system behavior, and generally high costs in terms of model identification and/or synthesis. Moreover, formal approaches, in particular classic control theory, are generally applicable under strong assumptions in terms of linearity, monotonicity, and reliability of the controlled system, thus significantly reducing the system viability zones. Under these assumptions, feedback-loop approaches can be proven robust to a certain degree of error in the estimation of the model parameters, hence they can provide effective control actions (for example, maintaining a performance metric within a certain range) even without self-tuning at runtime. However, both errors in parameter estimation and wrong assumptions on system behavior properties can consistently reduce the efficiency of the control in terms of possible SLO violations or resource assignment.

In the rest of this section, we provide representative examples for each class of static approaches that are and can be used inside Cloud IaaS controllers.

3.1 Approaches Based on Control Theory

Cloud controllers based on control theory adapt classic control theory techniques in the context of computing systems aiming to design adaptive, robust, and stable systems [17]. Here we provide only basic information about classic control theory. Interested readers can refer to the classic book by Hellerstein et al. for additional details on control theory approaches to managing computing systems [9].

Some approaches adapt standard control techniques, such as proportional, integral and derivative techniques, to synthesize self-adaptive controllers for the Cloud. These controllers rely on simple models and provide formal guarantees on the behavior of the system, but rely on very strong assumptions that are not verified in highly varying environments. Other approaches try to extend the scope of applicability of controllers based on classic control theory, by proposing complex parametric models where part of the parameters are unknown at design time. These approaches can still provide a limited set of formal guarantees depending on the assumption of proper on-line estimation methods.

An interesting example of approaches that apply classic control theory for designing Cloud controllers is the control theoretic solution that Maggio et al. propose to deal with self-optimization problems [16]. This solution is developed in the context of a common framework for monitoring system performances and adjust the allocation of resources to applications in order to guarantee a predefined service level.

Classic Control. Maggio et al. start with a simple (stateless and linear) model of the system that assumes a monotonic relation between allocated resources and application target performance. Then, they synthesize a Deadbeat controller, which is a common choice in the context of standard control theory, to track the target performance signal. They study the transient behavior of this controller by setting different parameters to estimate the approximation of the input signal, and to decide if the controller can effectively regulate the system despite its variations. When they cannot determine the suitability of the current model, they further refine the model by introducing more complex techniques and see if the new models are conclusive.

Advanced Control. Maggio et al. improve the basic deadbeat controller by adding an identification block that supports the online estimations of the unknown system parameters. They extend the controller by means of a Kalman filter and a Recursive Least Square algorithm. They observe that adopting more complex solutions, i.e., solutions that use more parameters to describe the relationship between the controlled and target variable, increases the difficulty to prove suitable control properties.

Multi Model Adaptive Control. A good example of multi model controllers is the approach of Patikirikorala et al. who use a particular instantiation of Multi-Model Switching and Tuning (MMST) adaptive control [20]. In a nutshell, they define several (fixed) linear models that describe the system behavior in different working conditions, and synthesize different single-model controllers following the classic control theory. The overall controller monitors the system variables, computes an error metric for each of the models, and enable only the single-model controller that correspond to the minimum predicted error.

Approaches based on control theory provide high guarantees in terms of assurance: stability and dynamic properties of the controller and controlled system can be proved in rigorous mathematical terms. Accurate system identification further increases the control reliability.

Classic single-model approaches require simpler proofs and provide a clear separation between system regions where the control offers formal guarantees (viability regions) and where not. Multi-model approaches are more flexible and are meant to cover wider viability regions. In these approaches, each controller behaves as in single-model approaches, but no formal guarantees are offered when the overall control logic switches them on and off.

The high assurance level of classic controllers is based on strict assumptions of the behavior of the controlled system that may not be always demonstrated in practice.

3.2 Threshold and Rule Based Approaches

Threshold Based. Threshold-based policies are popular in current industrial applications, as they are simple and intuitive to understand. They are applied by defining lower and upper limits on target metrics that, when crossed, trigger reconfiguration actions. Threshold-based controllers are used in many companies such as Amazon[1], RightScale[2] and Scalr[3]. In many controllers currently used in industrial applications, upper and lower bounds are defined by the customers referring to low level metrics, such as CPU usage. Customers define also the corresponding reconfiguration actions. Dutreilh et al. [7] experiment with static threshold-based policies that rely on high level *SLA* metrics, such as response time. They define control policies based on upper and lower thresholds, fixed amounts of virtual machines to be either allocated or deallocated, and pairs of "inertia" durations that support scaling up and scaling down periods. For instance, if the response time exceeds the upper bound, the controller allocates virtual machines, and inhibits itself for the corresponding inertia interval. If the response time drops below the lower bound, the controller deallocates virtual machines, and again inhibits itself for an inertia interval.

Rule Based. Rule based approaches extend threshold solutions by considering different types of events, and allowing rules to trigger other rules following the common ECA (event, condition, action) paradigm. An interesting rule based approach is *Claudia*, a rule-based controller for virtualized services proposed by Rodero-Merino et al. [22]. Claudia is more general than most rule based approaches, since it considers service life-cycle events and user defined variables, called Key performance indicators (KPIs), as well as business level metrics that holistically describe the status of a complete virtualized service and eventually triggers rules that reconfigure the system. Claudia combines three different types of rules, all defined by Cloud users: *scaling rules* that resemble traditional threshold based approaches and change the number of allocated virtual machines, *reconfiguration rules* that act at deployment time and choose the size and type of virtual machine to be deployed, and *business rules* that constrain the automatic scaling behavior with respect to running costs by limiting for instance the total number of running virtual machines. Business rules consider also Cloud federation concerns, for instance by *migrating* virtual machines from one Cloud provider to another. Claudia monitors the virtual service execution

[1] http://aws.amazon.com/autoscaling/
[2] http://www.rightscale.com/
[3] http://scalr.net/

and periodically tries to fire user defined rules that eventually trigger suitable control actions.

Rule and threshold based approaches can be verified, at least to some extent, formally. Verifying threshold based controllers is generally simpler than verifying rule based controllers, because threshold based controllers presents a less complex set of constraints, while full-fledged rule-based systems may have several conflicting and interdependent rules. Rules are commonly set manually under the assumption that they are able to capture main phenomena and characteristics of the system. They remain stable during the runtime and rely on the assumption that the system evolves only as predicted. Their event and condition clauses clearly identify the system viability zones that, however, remain fixed and may not are able to capture unplanned systems evolution.

3.3 Approaches Based on Analytic Models

Controllers based on analytic models use utility functions that combine system performance metrics and business considerations, like revenue and resource usage cost, to find desirable system configurations. They compute system performance metrics (typically the response time) through either different queuing models or analytical representation of queuing models. Analytical approaches differ each other mainly in terms of the chosen queuing formalism, the complexity and accuracy of the models, and the policy adopted to solve the optimization problem.

Single QN. Benanni and Menascé [3] combine queueing models and combinatorial search to dynamically allocate resources to application environments. They associate a local controller to each application environment and use queueing network models (open models for transactional systems, closed model for batch processing system) to predict the performance metrics of the application environment. Each local controller computes a utility measure by combining performance metrics, service level agreements and penalty functions, and sends the computed utility to a global controller that uses a combinatorial search algorithm to identify the final resources allocation of the entire data center, i.e., of all the application environments. While exploring the configuration space, the global controller interacts with the local controllers by suggesting potential new resource allocations, and the local controllers respond with updated values of local utilities. The queueing networks and their parameters are defined at design time.

Architecture Level Performance Model. Huber et al. [10] introduce an architecture level descriptive model, from which they derive performance models that are used by self-adaptive controllers to predict the system behavior. When the input workload changes, the control algorithm adds resources (virtual machines) to the system in order to eliminate all actual and predicted service level agreement violations, then the controller removes the resources that are underutilized. Huber et al. assume the availability of the architecture level models and estimate the parameters of the model at design time, for example, resource usages, routing/calling probabilities, service times, and possible usage scenarios and user classes. Once the model is in place, an automatic procedure transforms it into a predictive performance model, i.e., a queuing network, that is used on-demand by the control algorithm.

Mixed Queueing Networks. Bi et al. [4] use an mixed queueing model to simulate multi-tier applications and define a non-linear optimization problem over it to dynamically decide on the system at per-tier. The mixed model combines an M/M/c queuing model, for the *front-end*, with several M/M/1 queueing models for the *per-layer* virtual machines. The optimizer uses the model to calculate the number of resources to provision at the each tier according to the target end-to-end response time for the tier that is assumed to be agreed with customers.

Multiple QNs. Dejun et al. [6] address Web applications modeled as acyclic compositions of services using a what-if-analysis and negotiation among the composed services. Each service estimates its own performance variation in the case of changes of the allocated resource or workload, and pass the estimate up through the invocation tree to produce local decisions that are incrementally aggregated all the way up to a root controller. Dejun et al. model the performance of each single service as a M/M/n/PS queue, and consider performance variations for configuration changes of a single VM (+1 or -1 machine per service). The controller adjust the predictions by estimating the service time for the queuing networks at runtime using a feedback control loop: A threshold on the prediction error of the system performance triggers a new estimation of the service time by using the latest measured response time. Dejun et al. show the effectiveness of the proposed controller for different types of compositions, and claim that a service level agreement for the front-end service allows for finer control of the composed services than service level agreement thresholds on each component.

The approaches based on different queuing networks that we review above are based on the assumption that the chosen queuing model provides a sufficiently accurate representation of the considered system (plus its workload, processors, architecture, and bottlenecks) and that the system is inherently *stable*, that is, it does not present emerging or unpredicted behaviors. They also make several assumptions on the system behavior, its relevant components, and the statistical properties of the workload. Finally, all queuing models require a set of parameters that are estimated at design time with no provision for adjustment at runtime.

Simple models require to estimate few parameters, hence they are easier to configure, but may not be very accurate. Complex models can be more accurate with respect to the system structure, but require to estimate more parameters, and thus demand extra time, and effort. More complex models do not always result in higher accuracy, because of the simplifying assumptions that enable analytically solutions. Richer models (for instance, layered versus plain queuing networks, or mix-aware versus mixed oblivious solutions) typically offer more adaptability, that is more possibilities of closer adaptation to the system, albeit at the cost of higher complexity. In principle, they should be able to provide better assurance (at least in terms of a closer resembling representation of the modeled system); in practice the high number of parameters that need to be correctly estimated might attenuate the expected improvement.

3.4 Approaches Based on Black Box and Surrogate Models

The use of analytic modes is limited by the need of accurately defining a model structure and computing many parameters. Black box and surrogate models address this problem by generating the models from data gathered during system executions, and thus obtaining models that correspond to the system by construction. We will see in the next section that black box and surrogate models can be derived and adjusted at runtime, thus increasing the flexibility and adaptability of the controllers. Here we survey the main models that are proposed to be tuned before the systems deployment, typically at testing time and during model training.

Case Based and Clustering. Vasic et al. [30] use a workload signature based on Hardware Performance Counters (HPC) to cluster workloads, and associate the identified clusters to previously measured appropriate resource allocations. A proxy collects and computes workload signatures both in the training and control phase, by replicating a fraction of the entire application workload that is directed to a profiler for sampling and measurement. At runtime, a profiler computes the incoming workload signature, a classifier associates it to a class, and the controller applies the recorded resource allocation in a single control action. The controller uses a metric of the *certainty* of the association of the workload to a cluster as an indication of the need to trigger a new training of the classifier. The approach also measures the *interference* of other applications running in the same infrastructure, where interference is defined as the ratio of the performance achieved in production w.r.t. the performance for the same workload signature and resource assignment measured at training time. Substantially, the system works as a cache for previously observed combinations of workload signatures and resource allocations. In case of a cache *miss*, the default policy is to bring the controlled system to its maximum resource allocation to prevent service level objective breaches. Threats to the validity and assurance of the control come mainly from the choice and appropriateness of the metrics used to compute the signature and cluster the workloads. In their experiments, the authors report that the clustering typically results in only few workload classes while applications can normally have very large workload and configuration spaces.

Multiple Surrogates. Trushkowsky at al. [28] address the on-line reconfiguration of storage systems in response to workload changes under stringent performance requirements. The controller manages SCADS [2], a key-value store that offers eventual consistency and an easy partitioning of the key-value stores, hence natively supports replication and elasticity. Two specific issues make the problem hard: 1) to scale a data-intensive system, data items must be moved or copied across instances, impacting negatively on service performance, and 2) high percentiles of response time have much higher variance (and therefore are much harder to control) than the center of the distribution (average response time). The latter issue can cause oscillations in classical closed-loop control. Trushkowsky at al. tackle these issues by introducing a model predictive control: The controller refers to a model of the system and its current state to compute the optimal sequence of control actions, executes the first action of the sequence, and then recomputes the optimal sequence of control actions to chose the next actions. The system performance model coupled with workload statistics can predict whether a server is likely to meet its service level objectives. In the case of predicted violations, the controller either spawns new server instances or activates "hot" standby ones, and copies or migrates data bins using a copy-duration model for planning. Trushkowsky at al. builds the server performance models using statistical machine learning (SML) on data obtained through *off-line* controlled experimental runs with steady state workloads. They claim that simple changes in workload will not affect the accuracy of these models, that however can degrade over time if the application or the underlying data change consistently, for instance when an individual request returns more data than during training. In this case, the off-line models would have to be rebuilt in production. The approach uses a linear classification model with logistic regression to predict whether a given workload mix (get and put operations) and intensity are likely to cause service level objective violations. The copy-duration model is obtained off-line through the linear regression of samples gathered by running copy operations on servers under different workload rates.

Multiple Surrogate Models. Sharma et al. [23] present Kingfisher, a cost-aware system to scale elastic applications. Kingfisher relies on several models that capture capacity, costs and reaction time concerns, and a linear optimization to solve the cost- transition-time- aware control problems. Pricing models, elasticity mechanism models (migration, replication, etc. on the different platforms) and server capacity (for different resource allocations) are all obtained empirically at design time. The controller uses the models at runtime to solve the integer linear program that accounts for both infrastructure and transition costs and derives appropriate control actions. Kingfisher assumes that applications have a multi-tier software architecture where each tier has its own quality of service requirements that must be met by provisioning sufficient capacity. Moreover, Kingfisher assumes the availability of a perfect forecaster that is defined as forecaster that knows the workload in advance, as well as the perfect estimation of the per-tier peak-workloads. To solve the integer linear program in reasonable time and for not trivial applications, Kingfisher employs a greedy heuristic with a bounded worst case.

Black-box surrogate models provide an assurance level lower than analytic models, because the quality of these approaches depends on properties that are difficult to formalize and measure at design time. For example, the quality of such models depends largely on the amount and quality of the data collected from the system runs, their distribution in the input/output features space, the training/fitting procedures adopted. Being dependent only on data collected at design time, these models may not reflect accurately the real production environment.

Designers must assume that the drift between testing and staging environment is negligible. Moreover, they assume that the system behavior remains stable at runtime. In a sense, these approaches leverages black-box surrogate model because of they capability to learn relations between system variables that are unknown or too complex to estimate precisely; however, they do not leverage this inner capability outside the design time activity, thus limiting the adaptability of the controllers.

3.5 Approaches Based on Heterogenous Models

Some approaches address the limitations of the different techniques by suitably combining heterogeneous static techniques.

Regression. Lim et al. [15] combine a *cloud controller* that manages the compute infrastructure with an *application controller* that manages the resource assignments for each application to satisfy some a given performance goal. For the application controller, they propose to use a classical integral control to add and remove virtual machines based on average virtual machine CPU utilization. Integral control can be proved stable for a continuous control signal. Unfortunately the interface between the cloud and application controller consists of a coarse grained actuator, since one cannot add a fraction of a virtual machine, thus causing possible oscillations of the controlled system. To mitigate this effect, Lim et al. use a proportional thresholding technique, where higher and lower thresholds become smaller as system size increases, rather than a fixed target value for CPU utilization. They also use linear regression to model the relationship between the CPU utilization and the cluster size.

Surrogate Model Analytic and Heuristics. Jung et al. [12] focus on controllers that take into account the costs of system adaptation actions considering both the applications (for example the horizontal scaling) and the infrastructure (for example the live migration of virtual machines and virtual machine CPU allocation) concerns. Thus, they differ from most cloud providers that maintain a separation of concerns, hiding infrastructural control decisions from cloud clients.

The controller relies on an *estimator* that uses 1) automatic off-line experimentation to build a *cost table* quantifying performance degradation for each type of control action and workload, 2) layered queue networks (LQN) to predict the performance of each system configuration given a workload (LQN parameters are estimated offline), and 3) an ARMA filter to estimate the duration for which the current workload will remain stable (i.e., within a band B). The estimate of the duration of the stability of the workload is used to decide whether expensive (long term) control actions are worth or not. The controller searches through the graph of all possible control actions to find the sequence of control actions that maximizes a utility function that takes into account benefits and penalties expressed in terms of the application service level objectives, as well as the relative impact of all the control actions.

Approaches that combine heterogeneous models increase the adaptability with respect to the single static approaches, at the price of reducing the provided assurance level.

4 Dynamic Approaches and Runtime Assurance

We use the term dynamic approaches to indicate approaches based on models of the controlled system that are tuned at runtime, after system deployment. We refer to these models as dynamic models to stress the runtime nature of their construction. These approaches privilege runtime adaptability over assurance. They produce an accurate representation of the modeled system by characterizing the system behavior through metrics collected from the actual system execution. Typically, they build an initial version of the model at staging-time from a set of training samples, and then update the model continuously at runtime while the system is running. Thus, they can adapt to unforeseen changes and behaviors.

Different classes of dynamic approaches are characterized by the type of surrogate model they adopt, ranging from several forms of regression (for example, linear, quadratic, LOESS and Kriging) to machine learning techniques (for example, neural networks and reinforcement learning). Different model types imply different query and update strategies, to account for instance for the possibility of incrementally updating a model or the time and computational complexity of a complete re-training.

Dynamic approaches typically model either a single or a combination of system performance metrics as a function of some endogenous system properties that represent the system *configuration* in terms of both system characteristics, for example, number of allocated VM instances, CPU cores or threads, and exogenous factors, such as for instance the workload applied to the system or the influence of other services co-located in the same infrastructure. The approaches surveyed in this section often differ in terms of the considered metrics and the characterization of the workload features (for instance, workload intensity or service mix), but share some important features. They are highly flexible, and adapt easily to the measured system behavior, because of the ability to learn from and adapt to emerging behaviors, and this is extremely important for instance when the configuration or workload space is too vast for extensive exploration at staging time. They impose few requirements on the experience and knowledge of the modeled system functioning and behavior, because the samples required for training the models come from externally measurable system features.

The adaptive nature of dynamic approaches is both their main feature and their Achille's heal: The ability of constantly change and adapt makes them particularly well suited to highly dynamic systems and, at the same time, makes it hard to assure the quality of the resulting system, and to estimate the correct partition of model training effort between staging time (*bootstraping*) and on-line learning.

In the following, we survey the main approaches of this category distinguishing between approaches based on surrogate models and approaches based on machine learning.

4.1 Approaches Based on Surrogate Models

Surrogate models are build from sample executions of the modeled system, and describe the behavior of a system in terms of relations between input and output features that can be continuously updated with monitoring data. They are commonly used to describe either the steady-state behavior or the mid-to-long time horizon behavior projections. Self-adaptive controllers use surrogate models to predict the close future, given both the current and the estimated values of the input features. Some controllers use surrogate models also to support optimization procedures that explore the system configuration space to find the most suitable system configurations. Less commonly, surrogate models are used to provide model predictive control, where models are used to simulate and track the evolution of the system state under possible control actions, in order to plan for the most suitable ones. Such control strategy aims to maximize the control utility over a *receding* time horizon.

Splines and LOESS Regression. Bodik et al. [5] use statistical machine learning, and in particular smoothing splines and local regression (LOESS), to build performance models of the controlled system. Bodik et al.'s models represent response time as a function of workload intensity and system configuration. The controllers increase the robustness and the adaptivity to changes, by means of *model management* techniques (i.e., online training and change point detection) that update the model, track its quality and eventually rebuild it from scratch. The models are trained with data obtained online from the production environment. The controllers are conservative: They start from the maximum allowed allocation of resources, and decrease the allocation, while incrementally learning optimal configurations, to minimize service disruptions in exploration mode. Building models entirely from online samples simplifies the training phase, but may result in slow convergence of the models to new and possibly temporary system behaviors.

Kriging Models. In a recent work, we proposed an autonomic controller for horizontal scalability based on performance models of the controlled system. For each service level objective, the controller builds a different Kriging model that represents the objective metric (for instance the response time or the throughput) as a function of the number and types of virtual machine instances (system configuration) and a representation of the workload intensity and mix. Kriging models, also called Gaussian Process Regressions (GPRs), approximate target functions by means of a *spatial* correlation of samples. They extend traditional linear regression with a statistical framework that allows them to predict the value of the target function in un-sampled locations together with a confidence measure. In the Kriging based approach, training samples are collected by measuring the system behavior first at staging time to build an initial version of the models, and then in production to continuously updated them. As more samples are used, the accuracy of the model improves, while the uncertainty decreases, and the time to build the model increases. To avoid the collection of unmanageable sets of samples, many of which do not provide additional information to the model, the controller filters out old samples belonging to the same configuration. Kriging based approaches pair predictions with confidence measures that support the implementation of robust control policies (by tuning the desired risk of violations, impacting on assurance), and drive the exploration and exploitation decisions when learning the system behavior, thus improving over other regression mechanisms.

The assurance of surrogate-based self-adaptive controllers relates to (1) the ability of the models to accurately represent system behavior also in the presence of noisy and missing data, that is, when the quality of data interpolation and regression decreases, (2) the speed of convergence of the learning process, and (3) the accuracy of the quantification of the uncertainty of the model predictions. For example, models that are updated online and frequently, that do not need a large training set, and that can provide an accurate measure of confidence for their predictions, provide higher assurance than models that are updated infrequently and cannot provide any confidence interval for their predictions. Controllers that continuously monitor the quality (accuracy, prediction error, etc.) of the models, and account for completely rebuilding of the models whenever necessary can adapt faster to emerging system behaviors than controllers that merely update the models with new data.

4.2 Approaches Based on Machine Learning

Machine learning techniques are commonly divided in model-*based* and model-*free* techniques, depending on the use of models. Both classes of techniques are exploited to build self adaptive controllers. The most popular control solutions that refer to model based techniques use artificial neural networks (ANN), while popular model-free techniques use reinforcement learning (RL) and clustering applied to the discovery of control rules. In model based solutions, the accuracy of the results depend on crucial choices such as the model structure and the training data, thus no a priori guarantees can be enforced. In model-free solutions, the level of assurance depends on the learning rate, the instability/evolution pace of the controlled system and the size of the action-configuration space, which is proportional to the amount of possible control actions.

Artificial Neural Networks. Artificial neural networks use training samples to build a model of system dynamics that can predict the system reaction to different inputs. The structure of the network and the quality of the training data are critical to the performance [17] and must decided by the designers off-line. After the initial supervised training that sets all the internal parameters, artificial neural networks can be updated on-line by a back-propagation procedure based on punishment-reward concepts. Maggio et al. [16] implement a neural network to control the amount of resources allocated to process at the OS level that guarantees a given service level. Although the context is different from Cloud computing (where artificial neural network based controllers have not been used yet) the basic concepts of modeling and dynamically allocate resources are similar.

Reinforcement Learning. Controllers based on reinforcement learning learn directly optimal control policies, that is, the best set of actions to apply whenever the system enter a state, and thus do not require a model of the system. At runtime, reinforcement learning-based controllers adopt a trial-and-error learning strategy and apply (at least at the beginning) random actions. Effects on the system, and controllers utility function, resort either in action reward (if the actions increased the utility) or punishment (if the actions damages the system, thus lowering the utility). Reinforcement learning solutions suffer from poor scalability in the action-state space and from long convergence rates. To alleviate these limitations, Li and Venugopal [13] propose a distributed implementation of reinforcement learning based self-adaptive controllers. In this solution, each processing node, i.e, a virtual machine, is an independent entity and incorporates a local controller that runs a Q-Learning algorithm. Local updates to the model are pushed to the other controllers via a distributed hash table, so that collaborating controllers can learn the state-action-reward model quickly. At run time, each controller takes scaling decisions based on the shared model and aims to maximize the reward function while model updates are continuously published.

Clustering for Fuzzy Rules. Xu et al. [32] propose a dynamic approach based on fuzzy rules. The controller applies fuzzy modeling to learn the relationship between workload and resource demand, and uses clustering to update the rules at runtime. The controller is organized in two levels: at the lower level, local controllers are associated to physical resources and decide about the resource needs of the virtual applications deployed on the node; at the higher level, a global controller receives all the resource needs from the local controllers, and solves the resulting global optimization problem to decide the final resource allocation. Local controllers estimate the needs of resources at regular intervals for each virtualized application by means of fuzzy inferences: Each controller receives workload data, *fuzzifies* them, triggers the fuzzy rules, and finally produces the output *crispy*. At the same time, controllers analyze the monitoring data to derive new fuzzy rules, adapt existing ones, and remove not optimal rules from the knowledge base. Fuzzy rules are obtained by filtering and clustering raw monitoring data as they reach the controller: Data are filtered if they refer to mappings that lead to service level agreement violations, and then are clustered based on the density of surrounding data points. Eventually, a single rule is associated with each cluster. The controller defines an initial set of clusters, thus a set of fuzzy rules, offline using data from staging experiments, and updates the fuzzy rules at runtime, by adapting the size and number of clusters.

Control solutions based on machine learning techniques are in principle the most flexible solutions, since they can learn any kind of relations either directly modeling the system or capturing the optimal control policy. However, this extreme adaptability come at the cost of the impossibility of proving stability, convergence or any other properties important for control purposes. In particular, artificial neural network and reinforcement learning solutions do not provide any automatic means to evaluate the goodness of their fit nor their predictions, and designers have to either believe in them or not, and employ some external mechanism to track their error and retrain (whenever it is possible) the model. Machine learning solutions can be useful where the complexity of the controlled system makes it infeasible the construction of any satisfactory analytical solution or white box model, or when designers have no a-priori information about the behavior of the controlled system.

5 Hybrid Approaches and Combined Design and Runtime Assurance

We use the term hybrid approaches to indicate approaches that combine static and dynamic techniques to benefit from high adaptability while guaranteeing high assurance level. Static and dynamic approaches can be combined in many different ways, resulting in different blends of assurance and adaptability.

We distinguish two classes of hybrid approaches depending on the adopted merging strategy: (1) approaches that augment static solutions with learning and (self-)adapting capabilities, and (2) approaches that complement dynamic solutions with static models to modulate the effects of learning on the control behavior. Approaches that augment static technique with learning capabilities aim to increase the level of adaptability while maintaining a high assurance level. Approaches that combine dynamic solution with static model aim to improve assurance while limiting the loss along the adaptability dimension.

5.1 Static Approaches Augmented with Learning Capabilities

Static approaches augmented with learning and self-adapting capabilities increase the adaptability of the underlying static technique while maintaining high assurance level. They try to augment the size of viability zones by allowing the control solutions to deal with unseen and emerging behaviors that may differ from the design time assumptions. Augmented static solutions either relying on static models with online parameters tuning, or on static models that are rebuilt while the system is running.

Static Model with Online Parameter Tuning Approaches that augment static models with online parameter tuning are based on a core static model whose parameters are updated at runtime using monitoring data. The update and learning processes proceed in parallel with the controller activities.

LQN and Kalman Filter. Woodside, Zheng and Litoiu [31] propose a model-based feed-forward solution centered around a layered queueing network model of the system whose parameters are recomputed online. The controller uses the model to predict the system performance depending on the monitored workload, and to optimize the system configuration, in terms of server configurations and resources allocation. The controller adapts the parameters at runtime by means of an Extended Kalman Filter: The filter keeps updating the parameters until the residual error is below a threshold, and in this way the controller relies always on an accurate model of the system. The tracking filter greatly improves the robustness of the control algorithm in the presence of parameter drift. Moreover, extended Kalman filters have proven optimal properties when the relations between variables are linear, and are expected to have near-optimal properties if non-linearities are involved depending on the local properties around the operating point.

QN and Regression. Urgaonkar et al. [29] propose a dynamic capacity provisioning model for multi-tier Internet applications that uses queuing networks to determine the provisioning of resources for each tier of the application. Differently from the previous work, Urgaonkar et al. use online monitoring data to estimate the session arrival rate, the session duration and other parameters that are fed to the queuing network for the predictions. The proposed controller combines predictive and reactive methods to determine when to provision resources, to cater for respectively long-term / cyclic and short-term / unpredicted variations in the application workload. The controller computes long-term provisioning with the queuing network model where each tier is modeled as G/G/1 queue, and tiers are linked with replication factors to describe how the workload demand is distributed inside the controlled system. The controller deals with short-term variations of the load by means of a *sentry* component that implements admission control policies.

QN and Clustering. Singh et al. [24] use a queueing network to model the system as well, but they rely on mix-aware provisioning techniques to handle non-stationarity in workload mix and volumes. The controller employes k-means clustering to automatically classify the workload mix and uses the queuing model to predict the server capacity and support configuration optimization. The "workload class" is the parameter estimated online by the controller that keeps the model up to date. The initial clustering is computed off-line following an iterative and empirical process. On-line clusters are adjusted (split/merge) whenever the error of the estimated cluster predicted mean service time is greater than a given threshold with respect to the mean service time monitored by the mix-determiner. Maximum number and size of clusters are specified beforehand. Similar to the other approaches in this group, Singh et al. assume that the system can be modeled as a pipeline of independent tiers, for which per-tier service level agreement can be defined, and per-tier demands can be derived from the incoming one. The clustering allows to determine precisely the different types of requests in the workload improving the accuracy of results form the queuing model.

These approaches share the adoption of an analytic system representation (either queuing networks or layered queuing networks) whose structure remains unchanged. They achieve a limited degree of adaptability by estimating one or more parameters online. This caters for situations in which there is a need for minor adjustments with respect to design time expectations on system behavior, but the limits imposed by the initial queuing network model prevent the control from adapting to any possible emergent behavior related for instance to non-modeled system bottlenecks.

Model Update. Approaches that rebuild the model at runtime use a control model that is modified or completely rebuilt at system runtime, either periodically at a fixed time interval or whenever some indicative metric (for instance, a prediction error) crosses an acceptable threshold value.

Rule Base with Bayesian Classification. Jung et al. [11] propose a rule based approach, where rules are automatically generated by means of a machine learning process. The controller uses the rules as in static approaches: It monitors system variables, for instance, the workload, and triggers the control rules when necessary, e.g. when there is a match with the workload intensity, to change the system configuration. The controlled systems are modeled by means of layered queuing network models whose parameters are estimated offline, at design time. The controller then uses the layered queuing network models in a two step discovery process that is carried out at runtime in parallel with the control loop: (i) The controller randomly chooses a set of input workloads, searches for the corresponding optimal system configurations, and encodes the results as rules; (ii) It interpolates all the data via a Bayesian classification algorithm that derives a decision tree saved as policy. The decision tree covers the whole configuration space – thus not only the rules obtained in step (i) – and the learning algorithm can be configured to prune or merge, similar subtrees and configurations to simplify the rule set. The process produces a new decision tree with a finite number of leaves, i.e., system configurations, with a high degree of predictability and verifiability because the new tree encodes all the possible system configurations a priori enabling further decision to be taken at *business* level. The optimization procedure is based on a heuristic gradient search and considers both the system configuration, i.e. the number of replicas for each virtual machine, and their placement on a set of physical nodes. It assumes that the system utility monotonically decreases as resources are deallocated. The quality of the control, measured in terms of utility, depends on three critical factors: the model accuracy, the number of workloads considered during the optimization, and the "compactness" of the decision-tree, as more compact trees are less accurate.

QRS Model and Clustering. Quiroz et al. [21] propose a decentralized online clustering approach to detect patterns and trends in resource demands for jobs in Grid systems, and use this information to optimize the provisioning of virtual resources. The control is fully decentralized: In each control window, the clustering algorithm analyzes the incoming jobs and produces a number of target virtual machine classes. The number of virtual machines that must be provisioned is proportional to the volume of the incoming job for each class. Jobs are either assigned to the available virtual machines as they arrive or wait for the right virtual machine to start. Eventually, each local controller triggers the creation of new virtual machines to process the waiting jobs. Controllers rely on a model to estimate application service time based on Quadratic Response Surface Model (QRSM). The QRSM is fitted using long-term application performance monitoring data, and it is used at runtime to provide feedback about the quality of the clustering, i.e., the appropriateness of requested resources for the incoming jobs and their ability to meet QoS constraints. Given the actual resources, the QRS model is used to predict the response time for the estimated workload. This prediction is compared against the quality of service requirement and the controller uses the model to adjust the class attributes computed by the clustering. The controller then uses the quality measures obtained from the QRSM as an oracle to re-trigger the evaluation of the clusters with the decentralized online clustering algorithm.

Approaches that rebuild the model at runtime combine some static model, either a linear queuing network or a response surface model, with a learning/discovery mechanism that changes the control logic, rule- or clustering-based, respectively. The static model is derived from design time knowledge or long-term application historical data, and provides the fixed reference for the derivation/optimization and evaluation of the runtime-generated control logic. In a sense, the static model defines fixed boundaries for the controller behavior that make it predictable (i.e., within the boundaries), while the runtime adaptation aims at optimizing the control with respect to the measured system behavior.

5.2 Dynamic Approaches with Static Fall-Back

The lack of assurance of purely dynamic models derives from the dynamic nature of the data used to build and tune the models, that cannot be statically verified by definition. It is difficult to provide assurance proofs of controller behavior for approaches based mainly on runtime measurements, that heavily depend on the availability and quality of the data collected at runtime.

Dynamic approaches with static fall-back aim to improve the accuracy of these models by complementing the dynamic model with analytic approaches that might provide a reasonable alternative in the presence of low quality of the prediction based on the runtime model.

Case Based and Analytic. Malkowski et al. [18] propose a *multi-model* controller that combines the horizontal scale controller originally developed by Lim et al. [14] as the static approach with an *empirical model* obtained from runtime monitoring data. The empirical model uses a throughput vector space (i.e., a list of throughput values, one for each application interaction-type) to represent the combination of configuration, workload, and performances achieved by the system in a 30-seconds time frame. The empirical model is used to find the smallest (cheapest) system configuration that satisfies the service level objectives, and is located within a threshold value in terms of Euclidean distance in the throughput vector space with respect to the predicted workload. In other terms, it selects the smallest configuration among the set of *visited* configurations that were able to withstand a predicted workload intensity (or a comparably "close" intensity) without violating service level objectives. The controller switches to the static approach when the empirical model cannot find a visited configuration within the distance threshold.

Kriging and Analytic. In our recent work, we proposed a similar approach where the controller uses an analytical formulation, derived from a queue network model, as the static approach, and a Kriging model that interpolates monitoring data in [8]. Differently from the previous approach, we use an interpolation of data, i.e., the Kriging model, instead of raw monitoring data, to drive the control decisions, and use the same model to switch to the queuing network. In fact, the controller leverages the unique ability of Kriging models to provide a confidence measure along with performance prediction: If the confidence of the prediction is too low then the value is discarded and the controller resorts to the queuing network.

Reinforcement Learning with ANN and Queuing Networks. Tesauro et al. [26] employ both static and dynamic techniques in an approach that combines the strengths of reinforcement learning, artificial neural networks and queuing networks. A static technique is used when the reinforcement learning is in learning mode; in this period, the controller resorts to the queuing networks to control the system and collect training data at runtime. To speed up the learning, the data are used to train a non-linear function approximation, in this case an artificial neural network, of the Q-function that is used to obtain the learning rewards. Tesauro and co-authors represent the Q-function using neural networks instead of traditional look up tables to encode the state-action rewards. Artificial neural networks interpolate the collected samples and reduce the need of large training set, improving the controller scalability. By combining reinforcement learning and queuing networks, controllers can avoid poor performance during the training activities, because all the data are collected using the queuing network policy that provides an acceptable quality level of the control with respect to the random control actions of the reinforcement learning exploration. At the same time, controllers can improve their accuracy because steady-state queuing models are unable to take dynamical effects into account while the reinforcement learning can take into account dynamic effects such as transients and switching delays. Once the reinforcement learning is ready, the controller releases the queuing network and adopts it. In this period, parameters of the queue network are continuously updated based on measurements of system behavior, and when necessary, the controller can switch back to it and start the learning process again.

Hybrid approaches achieve high adaptability by using empirically obtained data to model emerging (and possibly unexpected) behavior such as for instance I/O bottlenecks (typically not modeled in analytic approaches / layered queuing networks for cloud controllers) or cross-layer interdependencies. High adaptability aims at a more precise representation of the system behavior geared towards the realization of *more efficient* (i.e., in terms of resource usage, service level objective violations) controllers. To compensate for the dynamic nature of black box models, and their dependency on possibly high varying runtime monitoring data, analytic approaches (typically queuing networks) are used as a safety net to constrain the controller actions. In a sense, analytic models provide the base controller behavior upon which dynamic solutions can improve.

6 Conclusions

In this chapter we presented the current state of the art of self-adaptive controllers for the Cloud Infrastructure as a Service. IaaS self-adaptive controllers dynamically assign resources to services aiming to strike the balance between under-provisioning (by minimizing service level agreement violations) and over-provisioning (by reducing resource assignments) in response to service workload variations. To efficiently allocate resources, self-adaptive controllers refer to some knowledge about the system characteristics and behavior that is typically encoded in terms of models or rules.

While designing self-adaptive controllers is a challenging task in itself, providing assurance guarantees on self-adaptive controllers behavior is even harder, because IaaS controllers typically face unpredictable environmental conditions (for instance, workloads and co-located services interference) and a very large space of configurations.

The different models used by the controllers come with different levels of assurance and adaptability. The inherent problem with adaptability is that it may give rise to undesirable emergent properties, impede the ability of administrators to understand system behaviors, and possibly reduce the predictability of the controlled system. However, controller adaptability may be required for instance in the case of systems too complex to be modeled analytically or whose environmental conditions and configuration spaces are too vast to be effectively explored before application deployment.

As it often happens, there is no "best" solution for IaaS controllers, the choice of the most suitable approach depends very much on the requirements posed on the control. We considered the trade-off between assurance and adaptability by classifying state-of-the-art approaches as belonging to either static (non-adaptable), dynamic (fully adaptable), or hybrid classes (partially adaptable). Static approaches privilege assurance over adaptability, being statically verifiable, but are less effective in dealing with situations not foreseen at design time. Dynamic approaches keep learning and adapting to the measured system behavior, hence offer models closer to the actual system behavior, typically resulting in a more precise (and efficient) control. However, this may come at the cost of reduced predictability. Hybrid approaches try to compensate the weakness of each of the previous classes by complementing static approaches with some degree of adaptability, and dynamic approaches with a fall-back static control.

We expect future research in this area to concentrate even more on hybrid approaches. The challenge is to be able to closely match system behavior, by continuously gathering performance measurements and adapting system models, while not completely giving up on formal proofs and guarantees on controller actions. To this end, some dynamic approaches, in particular the ones based only on the latest measured system behavior, might incur in the risk of learning the behavior of the system while in unperceived abnormal working conditions (for instance, false positives in monitoring VMs running and working state), thus building invalid models and consequently implementing the wrong control decisions. Hybrid approaches take advantage of the fall-back to less precise but typically more stable design-time models, and offer a safety net for situations in which the monitoring infrastructure cannot reveal malfunctions. Moreover, multi-model controllers can utilize possible discrepancies in the predictions from their different models to implement simple warning mechanisms (for example, requiring human intervention or interpretation of possible malfunctions) or various model-update policies, for instance with change-point detection. It is therefore our opinion that, independently of the levels of assurance or adaptability required by an application, multi-model solutions can typically offer a deeper insight on the controlled system behavior.

References

1. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Communication of ACM 53(4), 50–58 (2010)
2. Armbrust, M., Fox, A., Patterson, D., Lanham, N., Trushkowsky, B., Trutna, J., Oh, H.: Scads: Scale-independent storage for social computing applications. In: CIDR (2009)
3. Bennani, M.N., Menascé, D.A.: Resource allocation for autonomic data centers using analytic performance models. In: Proceedings of the International Conference on Automatic Computing, ICAC 2005, pp. 229–240 (2005)
4. Bi, J., Zhu, Z., Tian, R., Wang, Q.: Dynamic provisioning modeling for virtualized multi-tier applications in cloud data center. In: Proceedings of International Conference on Cloud Computing, CLOUD 2010, pp. 370–377 (July 2010)
5. Bodik, P., Griffith, R., Sutton, C., Fox, A., Jordan, M., Patterson, D.: Statistical machine learning makes automatic control practical for internet datacenters. In: Proceedings of the Conference on Hot Topics in Cloud Computing, HotCloud 2009, pp. 75–80 (2009)
6. Dejun, J., Guillaume, P., Chi-Hung, C.: Autonomous resource provisioning for multi-service web applications. In: Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, pp. 471–480 (2010)
7. Dutreilh, X., Rivierre, N., Moreau, A., Malenfant, J., Truck, I.: From data center resource allocation to control theory and back. In: Proceedings of International Conference on Cloud Computing, CLOUD 2010, pp. 410–417 (July 2010)
8. Gambi, A.: Kriging-based Self-Adaptive Controllers for the Cloud. PhD thesis, University of Lugano (2012)
9. Hellerstein, J., Diao, Y., Parekh, S., Tilbury, D.: Feedback Control of Computing Systems. Wiley (September 2004)
10. Huber, N., Brosig, F., Kounev, S.: Model-based Self-Adaptive Resource Allocation in Virtualized Environments. In: SEAMS 2011: 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Honolulu, HI, USA (2011); Acceptance Rate (Full Paper): 27% (21/76)
11. Jung, G., Joshi, K., Hiltunen, M., Schlichting, R., Pu, C.: Generating adaptation policies for multi-tier applications in consolidated server environments. In: Proocedings of the International Conference on Autonomic Computing and Communications, ICAC 2008, pp. 23–32 (June 2008)

12. Jung, G., Joshi, K.R., Hiltunen, M.A., Schlichting, R.D., Pu, C.: A Cost-Sensitive Adaptation Engine for Server Consolidation of Multitier Applications. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 163–183. Springer, Heidelberg (2009)
13. Li, H., Venugopal, S.: Using reinforcement learning for controlling an elastic web application hosting platform. In: Proceedings of the International Conference on Autonomic Computing, ICAC 2011, pp. 205–208 (2011)
14. Lim, H.C., Babu, S., Chase, J.S.: Automated control for elastic storage. In: Proceeding of the International Conference on Autonomic Computing, ICAC 2010, pp. 1–10 (2010)
15. Lim, H.C., Babu, S., Chase, J.S., Parekh, S.S.: Automated control in cloud computing: challenges and opportunities. In: Proceedings of the Workshop on Automated Control for Datacenters and Clouds, ACDC 2009, pp. 13–18 (2009)
16. Maggio, M., Hoffmann, H., Papadopoulos, A., Panerati, J., Santambrogio, M., Agarwal, A., Leva, A.: Comparison of decision making strategies for self-optimization in autonomic computing systems. ACM Transactions on Autonomous and Adaptive Systems (to appear)
17. Maggio, M., Hoffmann, H., Santambrogio, M.D., Agarwal, A., Leva, A.: Decision making in autonomic computing systems: comparison of approaches and techniques. In: Proceedings of the International Conference on Autonomic Computing, ICAC 2011, pp. 201–204 (2011)
18. Malkowski, S.J., Hedwig, M., Li, J., Pu, C., Neumann, D.: Automated control for elastic n-tier workloads based on empirical modeling. In: Proceedings of the 8th ACM International Conference on Autonomic Computing, ICAC 2011, pp. 131–140 (June 2011)
19. Menasce, D.A., Almeida, V.: Capacity Planning for Web Services: metrics, models, and methods. Prentice Hall (2001)
20. Patikirikorala, T., Colman, A., Han, J., Wang, L.: A multi-model framework to implement self-managing control systems for qos management. In: Proceeding of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, pp. 218–227 (2011)
21. Quiroz, A., Kim, H., Parashar, M., Gnanasambandam, N., Sharma, N.: Towards autonomic workload provisioning for enterprise grids and clouds. In: Proceedings of the IEEE/ACM International Conference on Grid Computing, GRID 2009, pp. 50–57 (2009)
22. Rodero-Merino, L., Gonzalez, L.M.V., Gil, V., Galán, F., Fontán, J., Montero, R.S., Llorente, I.M.: From infrastructure delivery to service management in clouds. Future Generation Computer Systems 26(8), 1226–1240 (2010)
23. Sharma, U., Shenoy, P., Sahu, S., Shaikh, A.: A cost-aware elasticity provisioning system for the cloud. In: Proceedings of International Conference on Distributed Computing Systems, ICDCS 2011, pp. 559–570 (June 2011)
24. Singh, R., Sharma, U., Cecchet, E., Shenoy, P.: Autonomic mix-aware provisioning for non-stationary data center workloads. In: Proceeding of the International Conference on Autonomic Computing, ICAC 2010, pp. 21–30 (2010)
25. Tamura, G., Villegas, N.M., Muller, H., Sousa, J.P., Becker, B., Karsai, G., Mankovskii, S., Pezze, M., Schafer, W., Tahvildari, L., Wong, K.: Towards practical run-time verification and validation of self-adaptive software systems. In: de Lemos, R., Giese, H., Müller, H., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems. Dagstuhl Seminar Proceedings (2011)

26. Tesauro, G., Jong, N.K., Das, R., Bennani, M.N.: On the use of hybrid reinforcement learning for autonomic resource allocation. Cluster Computing 10(3), 287–299 (2007)
27. 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)
28. Trushkowsky, B., Bodik, P., Fox, A., Franklin, M.J., Jordan, M.I., Patterson, D.A.: The scads director: scaling a distributed storage system under stringent performance requirements. In: Proceedings of the USENIX Conference on File and Stroage Technologies, FAST 2011, pp. 12–26 (2011)
29. Urgaonkar, B., Shenoy, P.J., Chandra, A., Goyal, P., Wood, T.: Agile dynamic provisioning of multi-tier internet applications. ACM Transactions on Autonomous and Adaptive Systems 3(1), 1–39 (2008)
30. Vasic, N., Novakovic, D., Miucin, S., Kostic, D., Bianchini, R.: DejaVu: Accelerating resource allocation in virtualized environments. In: Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, p. 13 (March 2012)
31. Woodside, M., Zheng, T., Litoiu, M.: Service system resource management based on a tracked layered performance model. In: Proocedings of the International Conference on Autonomic Computing and Communications, ICAC 2006, pp. 175–184 (2006)
32. Xu, J., Zhao, M., Fortes, J., Carpenter, R., Yousif, M.: On the use of fuzzy modeling in virtualized data center management. In: Proceeding of the International Conference on Autonomic Computing, ICAC 2007, pp. 25–35 (2007)

Author Index