Runtime Quantitative Verification of Self-Adaptive Systems

Simos Gerasimou

PhD

University of York

Computer Science

September 2016

Abstract

Software systems used in mission- and business-critical applications in domains including defence, healthcare and finance must comply with strict dependability, performance and other Quality-of-Service (QoS) requirements. Self-adaptive systems achieve this compliance under changing environmental conditions, evolving requirements and system failures by using closed-loop control to modify their behaviour and structure in response to these events.

Runtime quantitative verification (RQV) is a mathematically-based approach that implements the closed-loop control of self-adaptive systems. Using runtime observations of a system and its environment, RQV updates stochastic models whose formal analysis underpins the adaptation decisions made within the control loop. The approach can identify and, under certain conditions, predict violation of QoS requirements, and can drive self-adaptation in ways guaranteed to restore or maintain compliance with these requirements. Despite its merits, RQV has significant computation and memory overheads, which restrict its applicability to small systems and to adaptations affecting only the configuration parameters of the system.

In this thesis, we introduce RQV variants that improve the efficiency and scalability of the approach, and extend its applicability to larger and more complex self-adaptive software systems, and to adaptations that modify the structure of a system. First, we integrate RQV with established efficiency improvement techniques from other software engineering areas. We use *caching* of recent analysis results, *limited lookahead* to precompute suitable adaptations for potential future changes, and *nearly-optimal reconfiguration* to eliminate the need for an exhaustive analysis of the entire reconfiguration space. Second, we introduce an RQV variant that incorporates evolutionary algorithms into the RQV process facilitating the efficient search through large reconfiguration spaces and enabling adaptations that include structural changes. Third, we propose an RQV-driven approach that decentralises the control loops in distributed self-adaptive systems. Finally, we devise an RQV-based methodology for the engineering of trustworthy self-adaptive systems. We evaluate the proposed RQV variants using prototype self-adaptive systems from several application domains, including an embedded system for unmanned underwater vehicles and a foreign exchange service-based system. Our results, subject to the adaptation scenarios used in the evaluation, demonstrate the effectiveness and generality of the new RQV variants.

List of Contents

List of Figures

List of Tables

To my grandmother Chionou,
and my grandfather Simos

Acknowledgements

First, I am grateful to my supervisor Dr. Radu Calinescu. He has invested a great amount of time and effort in providing guidance throughout the duration of this research project. Without his support, this thesis would have never materialised.

Special thanks are due to my internal assessor Prof. Richard Paige for his invaluable feedback and continual encouragement, and Prof. Marin Litoiu for his suggestions to improve this thesis.

I would like to thank DSTL for supporting this project and Dr. Alec Banks, our DSTL technical partner, whose valuable insights in several aspects of our work have shaped and considerably improved this thesis.

I would also like to thank my colleagues and friends in the Enterprise Systems group, especially Dr. Babajide Ogunyomi, Dr. Yasmin Rafiq, Dr. Thomas Richardson, Gabriel Costa Silva, Dr. Colin Patterson, Dr. Konstantinos Barmpis, Dr. Antonio Garcia-Dominguez, Adolfo Sanchez-Barbudo Herrera, Dr. Ran Wei and Athanasios Zolotas for their support and for engaging in interesting discussions.

I had the opportunity to collaborate with great researchers and academics, especially Prof. Tim Kelly, Dr. Ibrahim Habli, Prof. Danny Weyns, Usman Iftikhar and Dr. Giordano Tamburrelli. Special thanks to you all.

I am very grateful to Christina for her love, encouragement and endless patience through the ups and downs of this journey. Thank you for your understanding and for being always by my side.

Finally, I would like to express my warmest gratitude to my family, Panagioti, Tasoula, Andri and Antoni, for their endless love and support, and for helping me to achieve my goals. Nothing would have ever been possible without you.

Declaration

Except where stated, all of the work contained in this thesis represents the original contribution of the author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Chapter 6 is joint work with Prof. Tim Kelly and Dr. Ibrahim Habli from University of York, UK and with Prof. Danny Weyns and Usman Iftikhar from Linnaeus University, Sweden. We clarify in Section 1.5 our contributions for this chapter.

Parts of the research described in this thesis have been previously published in:

- Simos Gerasimou, Radu Calinescu, Alec Banks. **Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration**. In 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14). pages 115-124, 2014.
- Radu Calinescu, Simos Gerasimou, Alec Banks. **Self-adaptive Software with Decentralised Control Loops**. In 18th International Conference on Fundamental Approaches to Software Engineering (FASE'15). pages 235-251, 2015. (Nominated for an ETAPS'15 best paper award).
- Simos Gerasimou, Giordano Tamburrelli, Radu Calinescu. **Search-Based Synthesis of Probabilistic Models for Quality-of-Service Software Engineering**. In 30th International Conference on Automated Software Engineering (ASE'15). pages 319-330, 2015.

Part of the research described in this thesis is currently under review in:

- Radu Calinescu, Simos Gerasimou, Ibrahim Habli, Usman Iftikhar, Tim Kelly, Danny Weyns. **Engineering Trustworthy Self-Adaptive Software**. Submitted to Transactions on Software Engineering (TSE).

Other publications related to this research:

- Radu Calinescu, Marco Autili, Javier Camara, Antinisca Di Marco, Simos Gerasimou, Paola Inverardi, Alexander Perucci, Nils Jansen, Joost-Pieter Katoen, Marta Kwiatkowska, Ole J. Mengshoel, Romina Spalazzese, Massimo Tivoli. **Synthesis and Verification of Self-aware Computing Systems**. Self-Aware Computing Systems. pages 337-373, 2017.
- Radu Calinescu, Simos Gerasimou, Kenneth Johnson, Colin Paterson. **Using Runtime Quantitative Verification to Provide Assurance Evidence for Self-Adaptive Software: Advances, Applications and Research Challenges**. Software Engineering for Self-Adaptive Systems 3. (in print)

Chapter 1

Introduction

> Realistically, such [autonomic] systems will be very difficult to build and will require significant exploration of new technologies and innovations. That's why we view this as a Grand Challenge for the entire IT industry.
>
> Paul Horn, Senior Vice President IBM research
> Autonomic Computing: IBM's Perspective on the State of Information Technology, 2001

1.1 Self-Adaptive Software Systems

The engineering of self-adaptive software systems is an emerging research area [45, 56]. As stated in the autonomic computing manifesto [91, 126, 139], software systems characterised by well-determined functionality that is fixed at design-time are replaced by systems resilient to changes and with adaptive capabilities [127, 183]. State-of-the-art systems have gone beyond simply monitoring themselves, and are increasingly anticipating changes and discovering deviations from the desired behaviour. Through a closed control loop [139], and with limited administrator involvement, self-adaptive systems can modify their behaviour and internal structure in response to changing environmental conditions, evolving requirements and internal changes [27].

Since the advent of autonomic computing [91, 126, 139], numerous research projects have focused on devising methodologies and frameworks for engineering self-adaptive software systems. Extensive surveys of this area can be found in [127, 143, 183]. Some of these projects enhance systems with self-adaptation capabilities using intelligent agents [25, 197], biologically-inspired mechanisms like flocking and foraging [195, 216], learning [175, 198] or control theoretical techniques [78, 140]. Other approaches reason about adaptation using architectural [19, 92, 169] or behavioural [15, 96] models, i.e., abstract system representations that include only features of interest and omit unnecessary structural and behavioural characteristics, and implementation details. Recent research advocates the use of quantitative verification at runtime to improve the dependability of self-adaptive software systems [29, 32, 68]. This thesis investigates self-adaptive systems whose reconfiguration is driven by runtime quantitative verification.

The opportunities for research and innovation in self-adaptive systems are remarkable. The second Horizon 2020 ICT call alone will provide €72M of funding for research on robotics and autonomous systems [70]. Horizon 2020 European research roadmaps and UK research strategy documents envisage that advances in autonomous and self-adaptive systems will have a tremendous impact on society, on business and the global economy [2, 3, 179]. The social benefits in sectors including environment, ambient-assisted living, energy and transport are expected to be significant, while the potential economic impact is estimated around €1 trillion by 2025 [156]. Autonomous and self-adaptive systems will enhance almost every aspect of our lives and contribute to safer transportation, efficient manufacturing, secure systems and improved healthcare [179]. In a recent report, the UK Aerospace, Aviation & Defence Knowledge Transfer Network [4] highlights that *"If delivered properly, telehealth systems can reduce mortality rates by up to 45% and reduce the need for admissions to hospital by 20%"*.

Driven by these opportunities, the research community has explored the use of self-adaptive systems in mission- and business-critical applications in domains ranging from defence and healthcare to finance and robotics [17, 30]. Systems deployed in these domains are expected to operate predictably and to comply with strict functional and Quality-of-Service (QoS) requirements. Failing to guarantee predictable behaviour and correct operation can have catastrophic results, including financial loss, environmental damage and harm to humans. To avoid such undesired events, self-adaptive software systems must be capable of verifying their compliance with system QoS requirements before adopting a reconfiguration plan, timely and without exhausting the available resources. If this is achieved, dependable system operation will be guaranteed.

1.2 Runtime Quantitative Verification Overview

Recent research advocates the use of formal methods at runtime as a means of rigorously driving the reconfiguration of self-adaptive systems and supporting dependable adaptation [36, 208]. *Runtime quantitative verification (RQV)* is a mathematically-based approach from this area that implements the closed-loop control of self-adaptive systems. The approach uses stochastic models of a self-adaptive system and its environment, and updates these models using runtime observations (of aspects of interest). The formal analysis of these models steers adaptation inside the control loop of the system [32]. RQV was introduced in [38, 68] and developed further in [33, 39, 76, 130, 154]. Successful applications include dynamic reconfiguration of telehealth service-based systems [39] and autonomous management of cloud-computing infrastructure [130].

RQV enables adaptation through continual verification of temporal logic formulae (capturing the QoS requirements of the self-adaptive system) over parametric stochastic models. These models comprise states and state transitions associated with relevant states and configurations of the system, and with possible transitions between states, respectively. The model parameters denote system and environment uncertainty (i.e., system or environment behaviour unknown at design-time or evolving at runtime), and control decisions in the form of alternative system configurations. For instance, given an unmanned underwater vehicle (UUV) equipped with on-board sensors and deployed in an environment monitoring mission [94], the approach can drive adaptation so that

- "At least 300 measurements of sufficient accuracy are taken by the active sensors for every 100m travelled by the UUV";
- "The energy consumption of the sensors does not exceed 400J for every 100m travelled by the UUV".

A self-adaptive software system employs RQV to reverify its compliance with QoS requirements after environmental, requirement and system changes. Through a continual monitoring process, the system and its environment are observed at runtime, and the up-to-date system and environment states are determined. These states are used to instantiate a concrete model from a family (i.e., parametric) of system models associated with the different scenarios the system might operate in. The chosen model is analysed exhaustively using quantitative verification, to identify and/or predict violations of QoS requirements such as response time, availability and cost. When requirement violations occur or are predicted, the quantitative verification results enable the synthesis of a verified reconfiguration plan. The execution of this plan ensures that the system continues to meet its QoS requirements despite the changes identified in the monitoring step.

1.3 Motivation and Research Hypothesis

Recent research established RQV as an approach suitable for augmenting software systems with self-adaptive capabilities and for continually verifying their QoS requirements [32, 38, 68, 146]. Notwithstanding its merits, RQV is a model checking approach, and thus is affected by the *state-explosion problem* [13, 48], where the size of the model increases exponentially with the size of the system. Hence, RQV cannot operate with the low computational and memory overheads required by many realistic self-adaptive software systems. Even when RQV can analyse a single system reconfiguration efficiently, the need to analyse all possible alternative reconfigurations after each change renders RQV infeasible with the exception of self-adaptive systems with small configuration spaces. These limitations call for efficient and scalable RQV techniques [32].

Existing research to improve RQV efficiency has introduced variants of the approach that exploit various aspects of the runtime verification process. These RQV variants, which we analyse in Section 2.2.3, are (i) *compositional*, by using assume-guarantee model checking to verify component-based systems one component at a time [37, 130, 151]; (ii) *incremental*, by establishing the current verification results from those obtained in previous verification runs [82, 130, 154]; and (iii) *pre-computation*-based, by transforming temporal logic formulae into easy-to-evaluate algebraic expressions [76, 79]. These RQV variants reduce the RQV overheads but they are only applicable to discrete-time models, can support only the simplest structural changes in the verified model, and make limiting assumptions (e.g., that the model can be partitioned into strongly connected components each of which is much smaller than the original model).

This thesis presents research that complements and addresses several limitations of the solutions summarised above by improving the efficiency and scalability of RQV, aiming to extend its applicability to larger and more complex critical self-adaptive systems. We adapt methods from other software engineering areas, employ evolutionary algorithms, and propose a decentralised RQV variant suitable for distributed self-adaptive systems. Finally, we devise an RQV-based methodology for engineering trustworthy self-adaptive systems. The hypothesis underlying the research in this thesis is as follows:

> *Given the representation of key aspects of a self-adaptive system as Markov models and a set of QoS requirements defined in suitable probabilistic temporal logics, efficient runtime quantitative verification techniques can provide guarantees that the system continues to satisfy its QoS requirements in the presence of changes, for much larger systems and with much lower overheads than the standard RQV approach.*

1.4 Research Scope and Assumptions

Before proceeding with the presentation of our research, we should set its scope. This will clarify the RQV challenges that the research focuses on and the contributions made by this thesis. Thus, we discuss other RQV-related challenges that are outside the scope of this thesis [17, 32, 95]. We also analyse the assumptions that underpin the self-adaptive systems used for evaluating the techniques and methodologies in Chapters 3–6.

First, RQV is affected by the traditional model checking challenges of deriving the stochastic models from the actual software systems and formalising QoS requirements in appropriate temporal logic formulae. Despite the importance of these challenges, they primarily concern design-time activities, and thus are outside the scope of this thesis. There is also significant research that addresses these challenges. For the former challenge, ProProST [104] can transform QoS requirements expressed in natural language into probabilistic temporal logic formulae. For the latter, stochastic system models can be developed by domain experts, by analysing the system and its log files [97], or by using model-to-model transformation techniques [16, 89, 102].

Second, the effectiveness of RQV depends on the accuracy of the models analysed by the technique. Models that do not capture the actual system and environment behaviour may lead to invalid reasoning about the compliance of a system with its QoS requirements, incorrect decisions and unnecessary or delayed adaptation. Developing initial accurate models for self-adaptive systems at design time requires significant expertise. Maintaining, however, these models in sync with changes occurring in the system and its environment at runtime calls for rigorous online parameter and model learning techniques. Such techniques can be found in [34, 39, 68, 97, 141]. This is an equally important challenge for RQV, but it is also outside the scope of this thesis.

Third, RQV uses stochastic models and the corresponding temporal logic formulae for modelling and verifying QoS requirements of a self-adaptive software system, e.g., reliability, performance and cost. Thus, this research deals with self-adaptive systems whose QoS requirements can be formalised as temporal logic formulae and verified over stochastic models (that capture the behaviour of the self-adaptive systems).

Fourth, the self-adaptive systems that we consider comprise a managed software system and a monitor-analyse-plan-execute (MAPE) [139] controller. We assume that the managed system is already available and its components can execute low-level commands as instructed by the controller during adaptation steps. The controller monitors the managed system and its environment, analyses stochastic models to identify deviations from the expected behaviour, synthesises an adaptation plan to restore compliance

with QoS requirements and executes this plan to adapt the managed system.

Fifth, this thesis does not consider self-adaptive systems that operate in a "closed world", i.e., systems in which changes can be anticipated fully beforehand. In this scenario, adaptation decisions for each possible system and environment state can be computed at design-time, and a rule-based system would be sufficient to realise adaptations at runtime. Also, self-adaptive systems with hard real-time requirements and systems in which reverification must occur several times per second are outside the scope of this thesis (since the time required by RQV to analyse alternative system configurations will unavoidably violate these timing requirements).

Finally, unless otherwise stated, we assume that all self-adaptive systems used in this thesis have a *failsafe configuration*. If no valid configuration can be found within the available time or without depleting the available resources, this (system-specific) failsafe configuration is automatically activated to minimise or prevent any further damage.

1.5 Thesis Contributions

The main contributions of the thesis, described in Chapters 3–6, are summarised below.

Efficient RQV Using Conventional Software Engineering Methods
We establish that caching, limited lookahead and nearly-optimal reconfiguration can improve the efficiency of RQV. With caching, recent verification results are kept for some time and are reused when the same environmental changes are encountered again. The technique is particularly effective when the changes are small and localised. Limited lookahead uses idle CPU cycles to pre-verify system states deemed likely to occur in the near future. When system changes arise, if the current system state has already been verified, it is sufficient to retrieve these verification results. Finally, nearly-optimal reconfiguration terminates early a reconfiguration step, provided that a valid configuration has been identified and a "near-optimality" criterion is met.

We evaluate these techniques using a simulator for self-adaptive UUVs. The findings provide evidence that all the techniques and their combinations improve the RQV response time in many realistic scenarios. For each technique, however, there are some trade-offs. Caching and limited lookahead require additional storage for retaining the verification results. Limited lookahead also needs extra CPU for the pre-verification process. On the other hand, nearly-optimal reconfiguration operates with much lower overheads than standard RQV, at the expense of selecting a sub-optimal configuration.

Improving RQV Efficiency Using Evolutionary Algorithms
We propose EvoChecker, a search-based approach that drives reconfiguration in self-adaptive systems using evolutionary algorithms. EvoChecker encodes possible system configurations within a probabilistic model template and specifies QoS requirements as constraints and optimisation objectives. We develop a human-in-the-loop EvoChecker that produces the Pareto optimal configurations and asks system experts to validate adaptation decisions. We also implement an automated EvoChecker that uses a strategy to archive verification results from recent reconfigurations, such that future reconfigurations are synthesised much faster by starting from the archived historical results.

We evaluate each EvoChecker variant in two case studies from different application domains and achieve significant reductions in RQV overheads. The human-in-the-loop EvoChecker generates Pareto optimal configurations and assists system experts with making informed adaptation decisions. The use of archive updating strategies in the automated EvoChecker improves the search and identifies effective configurations faster than strategies that do not make use of the archive.

Extending RQV to Decentralised Control Loops
We introduce DECIDE, the first RQV-based approach for the engineering of decentralised control loops in distributed self-adaptive systems. Each component of a DECIDE-based system carries out the following steps: (1) RQV-driven local capability analysis to establish a summary of possible component contributions towards realising the system-level QoS requirements; (2) receipt of peer QoS capability summaries; (3) decentralised selection of component contribution-level agreements (CLAs); and (4) RQV-based local control loop that guarantees the component's compliance with its CLA and local QoS requirements. Infrequently, a component is affected by major changes and executes steps (1)–(4) again; at all other times, each component runs only step (4) independently.

We evaluate DECIDE using a simulated embedded system from the UUV domain, showing its efficiency and effectiveness. DECIDE drives reconfiguration with overheads that are several orders of magnitude lower compared to centralised RQV-based control loops. DECIDE can also scale with insignificant increase in overheads to systems of much larger sizes. Finally, DECIDE-based systems withstand component failures and can continue operating when peer components fail completely (provided that system-level QoS requirements can be satisfied by the other components).

Engineering Trustworthy Self-Adaptive Software Systems
We explore the provision of assurances in self-adaptive systems. We also show how

verification results generated by RQV can be used as assurance evidence to confirm the correctness of adaptation decisions. To this end, we introduce ENTRUST, the first tool-supported methodology for the end-to-end engineering of trustworthy self-adaptive software systems. ENTRUST spans both design-time and runtime activities, and supports the development of formally verifiable controllers whose adaptation decisions are driven by RQV, the generation of design-time and runtime assurance evidence, and the runtime instantiation of an assurance argument pattern for self-adaptive systems.

To validate ENTRUST, we apply it to the development of two self-adaptive systems from different application domains, an embedded system from the UUV domain and a service-based system from the domain of foreign exchange. Next, through an empirical evaluation of the two generated self-adaptive systems, we confirm the correct execution of the ENTRUST controller and the validity of the generated assurance arguments. Finally, we examine the overheads incurred for the generation of assurance evidence and confirm that they are acceptable for small-to-medium systems. In larger systems, ENTRUST can use the efficient RQV variants we introduce in Chapters 3–5 or the complementary techniques for improving the RQV efficiency reviewed in Section 2.2.3.

ENTRUST is joint work with Prof. Tim Kelly and Dr. Ibrahim Habli from University of York, UK and with Prof. Danny Weyns and Usman Iftikhar from Linnaeus University, Sweden. The following ENTRUST components correspond to existing research of our collaborators that has been integrated within ENTRUST, or to components developed collaboratively. First, the formally verifiable models used for instantiating the ENTRUST controller are developed by specialising application-independent model templates, adapted from the recent work of de La Iglesia and Weyns [99]. We extended these templates with elements specific to the ENTRUST controller (e.g. probabilistic verification engine) and the class of managed systems handled by ENTRUST (e.g., fail-safe configuration). Second, the trusted MAPE virtual machine that executes the controller models at runtime is developed by Prof. Danny Weyns and Usman Iftikhar [128]. Finally, the assurance argument pattern has been developed in Goal Structuring Notation [105] in collaboration with Prof. Tim Kelly and Dr. Ibrahim Habli.

The Big Picture

The contributions made in this thesis focus on the application of RQV in self-adaptive systems and address the following key RQV challenges: (1) state-space explosion; and (2) managing large configuration spaces. To this end, the thesis introduces a set of RQV techniques that improve RQV efficiency and scalability, and extend the applicability of the technique to larger and more complex critical self-adaptive systems. More specifi-

cally, we reduce RQV overheads up to one order of magnitude by extending RQV with a set of conventional software engineering methods, i.e., caching, limited lookahead and nearly-optimal reconfiguration (Chapter 3). These methods require limited effort from an engineer, caching and limited lookahead are optimal and nearly-optimal reconfiguration is guaranteed to complete. With EvoChecker, we reduce RQV overheads by several orders of magnitude and make RQV applicable to systems with large state and configuration spaces (Chapter 4). As a metaheuristics-based technique, however, EvoChecker incorporates the drawbacks of these algorithms (e.g., no optimality guarantees, stagnation). It also requires more engineer effort to implement. With DECIDE, we enable the application of RQV to distributed self-adaptive systems with overheads several orders of magnitude lower compared to centralised RQV-based control loops(Chapter 5). This technique also requires some engineering effort. Finally, we introduce ENTRUST, a methodology for engineering trustworthy self-adaptive systems using dynamic assurance cases (Chapter 6). In this methodology, RQV is used for the verification of stochastic models of a self-adaptive system and for driving adaptation decisions. The verification results are used to update the assurance case at runtime. This is an engineer-driven methodology that spans both design-time and runtime.

Our contributions in Chapters 3–5 reduce the RQV overheads and improve the scalability of the technique. Applying these techniques in a self-adaptive system requires domain knowledge, i.e., an engineer should examine the type and characteristics of the system (e.g., centralised or distributed, medium or large configuration space) and determine the most applicable technique. Some of these techniques could be used in conjunction (e.g., caching and EvoChecker). On the other hand, the ENTRUST methodology (Chapter 6) could be applied for the engineering of new self-adaptive systems and our efficient RQV techniques could undertake the relevant analysis and verification tasks of the methodology.

Prototype Self-Adaptive Software Systems

We evaluate our new RQV veriants using three self-adaptive systems from different application domains: (1) a software-controlled dynamic power management system adapted from [174, 188] and described in Appendix B; (2) an embedded system from the unmanned underwater vehicle (UUV) domain introduced in Section 2.2.1.1; and (3) a service-based system from the domain of foreign exchange (FX) presented in Section 4.1.

The UUV and FX systems have been developed as part of this research. For each system, we describe its operation, define its behavioural and/or architectural aspects of interest and specify the relevant QoS requirements. We also devise Markov models that

describe the relevant behavioural aspects of each system and formalise its QoS requirements in a suitable variant of probabilistic temporal logic. Finally, we develop prototype system implementations and use them for evaluating experimentally our contributions.

Depending on the semantics of each proposed approach, we adjust the self-adaptive systems accordingly. For instance, DECIDE (Chapter 5) deals with distributed self-adaptive systems. Thus, we extend the single-UUV system to a multi-UUV system and introduce QoS requirements specific to the distributed version of the system.

We did not use recent case studies or exemplars from the domains of embedded and service-based systems, either because they were made available subsequently to our work (e.g., [65, 206]) or because they did not meet the scope of our research introduced in Section 1.4 (e.g., [201]). Finally, the choice of the UUV system is also driven by our plan to apply RQV to cyber-physical systems in the future (cf. Chapter 7).

1.6 Thesis Structure

The remainder of this thesis is structured as follows.

Chapter 2 presents background information, an overview of RQV, and a survey of related work. Sections 2.1.1 and 2.1.2 introduce the Markov models and temporal logic variants used throughout the thesis, respectively. Section 2.2 reviews RQV, starting with the principles underpinning its use in self-adaptive systems (Section 2.2.1) and including an extensive analysis of recent research in the area (Sections 2.2.2 and 2.2.3).

Chapter 3 introduces our work on improving RQV efficiency using conventional software engineering techniques. Sections 3.1.1–3.1.3 present the integration of caching, limited lookahead and nearly-optimal reconfiguration with RQV. Section 3.2 describes the development of these RQV variants within the open-source platform MOOS-IvP for the implementation of autonomous applications on unmanned marine vehicles [20]. Section 3.3 presents our experimental evaluation and analyses our findings. Sections 3.4 and 3.5 discuss chapter-specific related work and conclude the chapter, respectively.

Chapter 4 describes EvoChecker, our search-based approach that uses evolutionary algorithms within the RQV process. Section 4.1 presents EvoChecker, including its modelling language (Section 4.1.1), and its human-in-the-loop (Section 4.1.3) and automated (Section 4.1.4) variants. Section 4.2 describes the open-source EvoChecker tool. Section 4.3 presents the evaluation of both EvoChecker variants and summarises our results.

Sections 4.4 and 4.5 review related work and summarise the chapter, respectively.

Chapter 5 introduces our DECIDE framework for the decentralisation of the control loops of distributed self-adaptive software. Section 5.1 presents DECIDE, and Sections 5.1.2– 5.1.6 describe its four steps. Section 5.2 explains the DECIDE implementation within the open-source platform MOOS-IvP. Section 5.3 reports the experimental evaluation carried out to assess the effectiveness and scalability of DECIDE. Sections 5.4 and 5.5 discuss chapter-specific related work and conclude DECIDE, respectively.

Chapter 6 introduces our ENTRUST methodology for the engineering of trustworthy self-adaptive systems. Section 6.1 presents ENTRUST, while details of its various stages are provided in Sections 6.1.1–6.1.6. Section 6.2 outlines the tool-supported ENTRUST implementation. Section 6.3 reports the evaluation performed to examine the generality, correctness and efficiency of ENTRUST. Sections 6.4 and 6.5 overview chapter-specific related work and summarise the chapter, respectively.

Chapter 7 concludes the thesis by summarising the findings and contributions of this research, and providing directions for future work.

Chapter 2

Background and Field Review

Designing and analysing software systems requires techniques that model and reason about their behaviour, considering specific aspects of interest while abstracting away implementation details. In this chapter, we present quantitative verification, a technique that is particularly suitable for this purpose. Section 2.1 provides basic background information, terminology and notation used throughout the thesis. In Sections 2.1.1 and 2.1.2, we present the types of probabilistic models and specification formalisms used in quantitative verification, respectively. In Section 2.2, we survey the area of runtime quantitative verification. More specifically, in Section 2.2.1 we advocate the use of runtime quantitative verification to support self-adaptation, while in Sections 2.2.2 and 2.2.3 we summarise early applications of the technique and recent advances aimed at improving its efficiency, respectively.

2.1 Quantitative Verification

Many software systems are subject to events of stochastic nature including message loss and component failure. **Probability** is an important element in defining these stochastic events and, therefore, in designing, analysing and verifying software systems [182]. Probability can be employed to:

- **Model system uncertainty**. Modern systems are typically deployed in dynamic and unpredictable environments. Moreover, self-adaptive systems are expected to operate dependably and to reconfigure themselves in response to unexpected changes. These uncertainty aspects can be described probabilistically.

- **Derive efficient algorithms**. Randomisation, e.g., in the form of electronic coin tosses, can break symmetry in distributed co-ordination algorithms. For instance, randomisation in self-stabilising algorithms [150] ensures that all processes will eventually reach consensus with probability 1. In the IPv4 Zeroconf Protocol [147], randomisation minimises IP address collision by requiring devices joining a network to choose randomly an address from a pool of 65024 available addresses.
- **Model system failure**. Hardware and software components are both error-prone and affected by multiple types of failures. Typical examples include hard disk damage, processor overheat and service degradation. Probability can be used to define this unreliable behaviour, to reason about the likelihood of an action to terminate incorrectly or to determine if system failure exceeds a certain threshold. An example statement for a fault-tolerant system is "the likelihood of a failure occurring within the first hour is at most 0.001".
- **Model system performance**. Probability enables to quantify the performance of a system and to establish various Quality-of-Service (QoS) properties such as throughput, average response time, queue length, energy consumption and CPU utilisation. This performance evaluation supports the identification of flaws in system design and potential deviation from specification, and can provide insights into improving the design. A typical QoS property for a server handling requests is "the expected cost of providing an answer does not exceed 10 time units".

Quantitative verification (QV) [32, 145] is a mathematically-based technique for analysing the reliability, performance and other QoS properties of systems exhibiting stochastic behaviour. The technique uses finite state-transition Markov models (cf. Def. 2.1) to describe the behaviour of a system. In these models, states represent different system configurations and edges correspond to available transitions between these states. Depending on the model type, an edge is annotated with the probability or rate of taking the associated transition. Also, model states and transitions can be augmented with cost/reward information, extending the range of properties that can be analysed.

The QoS properties analysed using these models are formally specified in variants of temporal logic, extended with probabilities and costs/rewards. These specifications enable reasoning about the likelihood of certain events occurring while a system operates or about the cost/reward associated with these events. Typical examples of QoS properties [182] include, "the expected time until a new device entering the network gets a reply" for the IPv4 Zeroconf Protocol and "the resource usage during the first month of operation" for a fault-tolerant system.

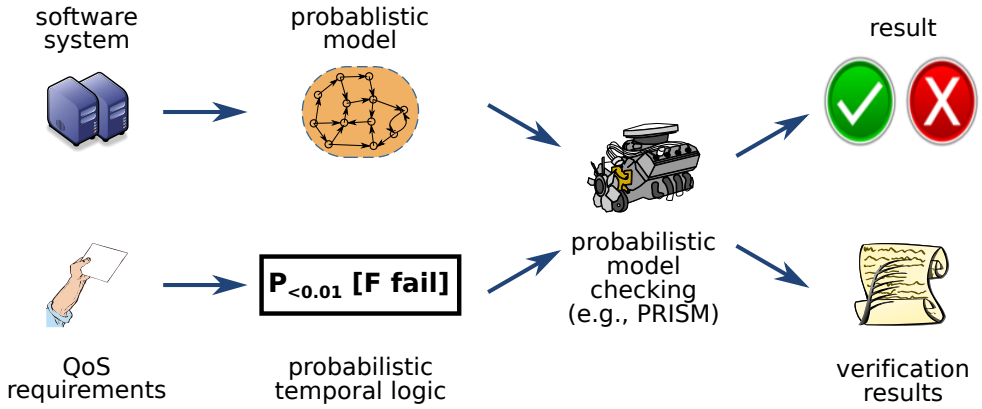


Figure 2.1: Overview of quantitative verification process.

Given a model and a probabilistic temporal logic formula, QV carries out exhaustive analysis to establish the value of the property. To this end, the technique can determine if the property meets a bound (or threshold), and thus answer to questions concerning the satisfaction of a QoS requirement. Alternatively, verification results in the form of exact probabilities or rewards associated with the formula can be used to gain insights into the behaviour of the system and support decision making. A pictorial overview of the technique is given in Figure 2.1. Methods such as symbolic model checking, symmetry reduction and counterexamples are used for reachability analysis [13], while numerical methods like linear algebra and linear programming are used to calculate the actual probabilities [182]. Probabilistic model checkers automate the process of quantitative verification. PRISM [149], MRMC [133] and Ymer [214] are among the most widely used probabilistic model checkers. We refer the interested reader to [1] for a comprehensive list of available model checkers.

In the remainder of this section, we introduce the Markov models used later in the thesis (Section 2.1.1) and the probabilistic temporal logics (Section 2.1.2) underpinning the approaches developed by this thesis.

2.1.1 Markov Models

Given that specific aspects of a software system's behaviour can be described probabilistically, e.g., uncertainty originating from the deployed environment or users' interaction, we can model this behaviour as a stochastic process, i.e., a collection of random variables indexed by time. Formally, a stochastic process is a function $X : T \times \Omega \to S$, where T is a set of time points, Ω is the sample space and S is the state space of X [180]. If

time evolves in discrete intervals, i.e., $T = \mathbb{N}$, X is a *discrete-time process*; otherwise, i.e., $T = [0, \infty)$, X is a *continuous-time process*. We denote with $X(t)$ the state of the process at time $t \in T$.

A special class of stochastic processes, called Markov processes, satisfies the Markov property (Def. 2.1), which specifies the conditional dependence of future on the present and its independence from the past [182]. Simply stated, evolution in a Markov process depends only on the present state and not on the history of preceding events.

Definition 2.1. *A stochastic process* $\{X(n) \mid n = 0, 1, 2, ...\}$ *satisfies the Markov property if*

$$P[X(n) = s_n \mid X(n-1) = s_{n-1},\ ...,\ X(0) = s_0] \quad = \quad P[X(n) = s_n \mid X(n-1) = s_{n-1}]$$

where $s_0, s_1, ..., s_k$ *represent successive states of the stochastic process.*

All the models described hereafter are variants of Markov processes. Depending on how time evolves and the aspects of system behaviour modelled, these variants could be discrete-time Markov chains (DTMCs) and continuous-time Markov chains (CTMCs).

2.1.1.1 Discrete-Time Markov Chains

Discrete-time Markov chains (DTMCs) are the simplest among the probabilistic models we consider. A DTMC is essentially a state-transition system augmented with probabilities in which time progresses in discrete intervals. The next state at each point in time is specified by a discrete probabilistic distribution from source to target states.

Definition 2.2. *A discrete-time Markov chain (DTMC) over a set of atomic propositions AP is a tuple*

$$D = (S, s_0, \boldsymbol{P}, L) \tag{2.1}$$

where:

- S *is a finite set of states;*
- $s_0 \in S$ *is the initial state;*
- $\boldsymbol{P} : S \times S \to [0, 1]$ *is a transition probability matrix such that the exit probability from any state* $s \in S$ *is* $\sum_{s' \in S} P(s, s') = 1$*;*

- $L : S \to 2^{AP}$ *is a labelling function which assigns to each state* $s \in S$ *the set* $L(s) \subseteq AP$ *of atomic propositions that are valid in that state.*

An *atomic proposition* declares a relevant characteristic of the system under investigation. In particular, an atomic proposition expresses simple statements associated with system states that evaluate to true, if they hold, or false, otherwise. Typical examples of atomic propositions are "the server queue is not full", "the system invoked its self-protection mechanism", and "the energy usage exceeds 100 units".

For any states $s, s' \in S$, the entry $\mathbf{P}(s, s')$ of the transition probability matrix $\mathbf{P}$ specifies the probability of moving from s to s' in a single step. For state s, row $\mathbf{P}(s, \cdot)$ and column $\mathbf{P}(\cdot, s)$ denote the transitions leaving from and entering to state s, respectively. A state with only an outgoing transition to itself with probability 1 is called an *absorbing state*. An edge from s to s' exists if and only if $\mathbf{P}(s, s') > 0$. Figure 2.2 shows the graphical representation of a DTMC, in which vertices and edges represent system states and transitions, respectively. The transition probability matrix $\mathbf{P}$ is shown in Example 2.1.

A *path* signifies a single execution of a DTMC. Formally, a path is a non-empty sequence of states $\pi = s_0 s_1 s_2 ...$ where $s_i \in S$ and $\mathbf{P}(s_i, s_{i+1}) > 0$ for all $i \geq 0$. A path is *finite* if the number of states in the sequence is finite, and has a length denoted as $|\pi|$. The i-th state of the path is denoted by $\pi(i)$. Given a starting state s, the set of all paths starting from s, is denoted $Path^D(s)$.

In order to analyse the behaviour of a DTMC, we need to evaluate the probability of taking certain paths through the model. The matrix $\mathbf{P}$ induces a probability space on $Path^D(s)$, using the cylinder construction [182]. An observation of a finite path determines a basic event (cylinder). Assuming a starting state s_0 and a finite path $\pi = s_0 s_1 ... s_n$, the probability $\mathbf{P}r_s(\pi)$ of reaching state s_n is defined as

$$\mathbf{P}r_s(\pi) = \begin{cases} 1 & \text{if } n = 0 \\ \mathbf{P}(s_0, s_1) \cdot \mathbf{P}(s_1, s_2) \cdot ... \cdot \mathbf{P}(s_{n-1}, s_n) & \text{otherwise} \end{cases} \tag{2.2}$$

This equation provides the means to find a unique probability $\mathbf{P}r_s$ on the infinite paths $Path^D(s)$. Generally, for any pair of states $s, s' \in S$, the probability $\mathbf{P}r_s$ is equal to the sum of the probabilities of all the paths starting from s and ending to s'. When the number of steps is limited to k, the probability is equal to the entry $\mathbf{P}^k(s, s')$. More details can be found in [13, 182].

Extending DTMCs with Rewards

DMTCs can be augmented with cost/reward structures, i.e., functions that map states and/or transitions to real-valued quantities. Although mathematically there is no distinction between manipulating and computing costs and rewards, commonly adopted semantics declare that cost should be minimised and reward should be maximised. These structures can be used to represent additional information regarding the behaviour of a system modelled by a DTMC. They can have a wide range of interpretations, for example, elapsed time, power consumption, size of message queue, numbers of messages successfully delivered, net profit and throughput.

Definition 2.3. *A cost/reward structure over a DTMC $D = (S, s_0, \boldsymbol{P}, L)$ is a pair of real-valued functions $(\underline{\rho}, \iota)$ where:*

- $\underline{\rho} : S \to \mathbb{R}_{\geq 0}$ *is a state reward function that defines the reward obtained when D is in state s for one time step;*
- $\iota : S \times S \to \mathbb{R}_{\geq 0}$ *is a transition reward function that defines the reward obtained each time a transition occurs.*

Example 2.1. Figure 2.2 depicts the DTMC model of a train booking e-commerce system. A customer, after entering the system, can search for and buy train tickets, and optionally have the tickets delivered using a shipping service, or can select to print the tickets herself. The customer can then either perform another search or complete the operation. The model elements are: the set of states $S = \{s_0, s_1, ..., s_7\}$, the initial state s_0, the set of atomic propositions $AP = \{\text{init}, \text{search}, \text{buy}, \text{shippping}, \text{succ}, \text{failed\_search}, \text{failed\_buy}, \text{failed\_shipping}\}$, and the labelling function L : $L(s_0) = \{\text{init}\}$, $L(s_1) = \{\text{search}\}$, $L(s_2) = \{\text{buy}\}$, $L(s_3) = \{\text{shipping}\}$, $L(s_4) = \{\text{succ}\}$, $L(s_5) = \{\text{failed\_search}\}$, $L(s_6) = \{\text{failed\_buy}\}$, $L(s_7) = \{\text{failed\_shipping}\}$.

The DTMC is augmented with a state cost structure, shown in the rectangular box, that associates a cost of 1 each time a search operation is made.

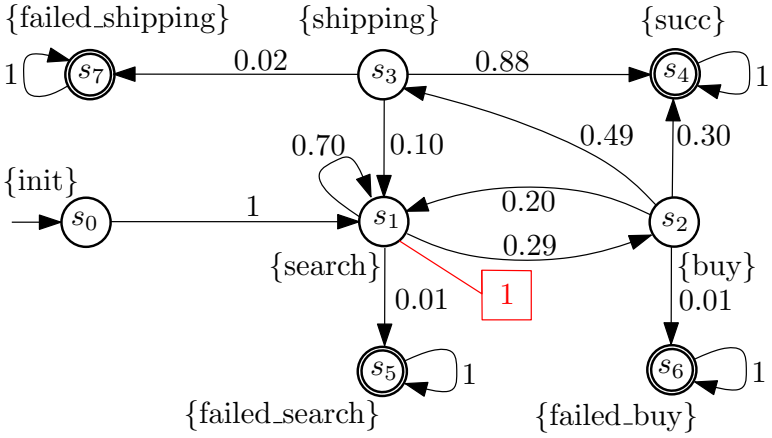


Figure 2.2: DTMC model of an e-commerce system.

The corresponding transition probability matrix $\mathbf{P}$ of the system is

$$\mathbf{P} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.70 & 0.29 & 0 & 0 & 0.01 & 0 & 0 \\ 0 & 0.20 & 0 & 0.79 & 0 & 0 & 0.01 & 0 \\ 0 & 0.10 & 0 & 0 & 0.88 & 0 & 0 & 0.02 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Suppose we want to estimate the probability to complete a purchase (i.e., reach state s_4) within $k = 3$ steps. The expected result is given by entry $\mathbf{P}^3(s_0, s_4)$. Since only path $\pi = s_0 s_1 s_2 s_4$ can reach the success state within at most $k = 3$ steps, the probability $\mathbf{P}r_s$ is equal to the probability given by path π, i.e., $\mathbf{P}r_s = \mathbf{P}r_s(\pi) = \mathbf{P}(s_0, s_1) \cdot \mathbf{P}(s_1, s_2) \cdot P(s_2, s_4) = 0.087$.

2.1.1.2 Continuous-Time Markov Chains

Continuous-Time Markov Chains (CTMCs) model systems which have discrete states but where time progresses continuously. Transitions in a CTMC occur in real time, while delays before making a transition are represented with exponential probability distributions [12, 148]. CTMCs are suitable for modelling and analysing the performance and reliability of real-time systems, including *transient* and *steady-state behaviour*, i.e., the state of the system at a particular time instant and in the long-run, respectively. Typical examples of such systems are queueing networks, financial systems and biological systems.

Definition 2.4. *A continuous-time Markov chain (CTMC) over a set of atomic propositions AP is a tuple*

$$C = (S, s_0, \boldsymbol{R}, L) \tag{2.3}$$

where:

- *S is a finite set of states;*
- *$s_0 \in S$ is the initial state;*
- *$\boldsymbol{R} : S \times S \to \mathbb{R}_{\geq 0}$ is a transition rate matrix;*
- *$L : S \to 2^{AP}$ is a labelling function which assigns to each state $s \in S$ the set $L(s) \subseteq AP$ of atomic propositions that are valid in that state.*

For any states $s, s' \in S$ such that $\mathbf{R}(s, s') > 0$, the probability that the CTMC transitions from state s to state s' within t time units is given by the negative exponential distribution $1 - e^{-R(s,s')\cdot t}$. When multiple outgoing transitions can be triggered in s, that is, $|T(s)| > 1$ where $T(s) = \{s' \in S \mid R(s, s') > 0\}$, the behaviour of a CTMC is the result of a race condition. Each outgoing transition from s can be triggered after an exponentially distributed delay. The first triggered transition determines the next model state. Before a transition occurs, the system remains in state s for time t, which is exponentially distributed with its *exit rate*

$$E(s) = \sum_{s' \in S} R(s, s') \tag{2.4}$$

When leaving state s, the probability to move to state s' is given by the *embedded DTMC of CTMC*, specified in Def. 2.5. The probabilities in the embedded DTMC are independent of the time at which the transitions occur. Similarly to DTMCs, this information can be used to determine the probability for the model to be in a given state after taking n transitions.

Definition 2.5. *The embedded DTMC of a CTMC* $C = (S, s_0, \boldsymbol{R}, L)$ *is a tuple*

$$emb(C) = (S, s_0, \boldsymbol{P}^{emb(C)}, L) \tag{2.5}$$

where for any $s, s' \in S$

$$\boldsymbol{P}^{emb(C)}(s,s') = \begin{cases} R(s,s')/E(s) & if E(s) \neq 0 \\ 1 & if E(s) = 0 \text{ and } s = s' \\ 0 & otherwise \end{cases}$$

A *path* through a CTMC is a non-empty sequence $s_0 t_0 s_1 t_1 s_2, \ldots, t_{k-1} s_k$ where $\mathbf{R}(s_i, s_{i+1}) > 0$ and $t_i \in \mathbb{R}_{>0}$ is the time spent in state s, for all $0 \leq i \leq k$. State s_k is absorbing. We denote with $\pi(i)$ the i-th state of path π. If $i \leq k$ and $t \leq \sum_{i=1}^{k-1} t_i$ we also denote with $time(\pi, i)$ the amount of time spent in state s_i and with $\pi@t$ the state occupied at time t; otherwise $time(\pi, i) = \infty$ and $\pi@t = s_k$. We use the notation $Path^C(s)$ for the set of all finite paths originating from state s.

Analysing the behaviour of a CTMC model requires to reason about the probability of traversing certain paths in the model. Given a set of non-empty intervals $I_0, ..., I_{n-1} \in \mathbb{R}_{\geq 0}$, the cylinder set $C(s_0, I_0, ..., I_{n-1}, s_n)$ contains all paths $\pi \in Path^C(s)$ such that $\pi(i) = s_i$ for all $i \leq n$ and $time(\pi, i) \in I_i$ for all $i < n$. The probability of this cylinder is defined by

$$\begin{aligned} &\mathbf{P}r_s(C(s_0, I_0, ..., I_{n-1}, s_n)) = \\ &\mathbf{P}r_s(C(s_0, I_0, ..., I_{n-2}, s_{n-1})) \cdot \mathbf{P}^{emb(C)}(s_{n-1}, s_n) \cdot \left(e^{E(s_{n-1}) \cdot inf I_{n-1}} - e^{E(s_{n-1}) \cdot sup I_{n-1}}\right) \end{aligned} \tag{2.6}$$

where $inf I_{n-1}$ and $sup I_{n-1}$ are the infimum and supremum of the interval I_{n-1}, respectively [148].

The sum of all possible cylinder sets involving states $s_0, s_1, ..., s_n$ and intervals $I_0, ..., I_{n-1} \in \mathbb{R}_{\geq 0}$ gives a unique probability $\mathbf{P}r_s$ for reaching state s_n.

Extending CTMCs with Rewards

CTMCs can be annotated with cost/reward structures of the form (ρ, ι), i.e., functions that assign real-valued quantities to states and transitions. Differently from DTMCs, state cost/rewards are calculated based on the rate at which they are obtained. Thus, if the model remains in state s for $t \in \mathbb{R}_{\geq 0}$ time units, a reward of $t \cdot \rho(s)$ is acquired.

Definition 2.6. *A cost/reward structure over a CTMC is a pair of real-valued functions* $(\underline{\rho}, \iota)$ *where:*

- $\underline{\rho} : S \to \mathbb{R}_{\geq 0}$ *is a state reward function that defines the rate at which the reward is obtained while the CTMC is in state* s*;*
- $\iota : S \times S \to \mathbb{R}_{\geq 0}$ *is a transition reward function that defines the reward obtained each time a transition occurs.*

Example 2.2. Consider the i-th sensor of a system equipped with a set of sensors that can measure an environment attribute, e.g., temperature, humidity or light intensity. When used, the sensor takes measurements with rate r_i and consumes energy e_i for each measurement. Once a measurement is taken, the sensor carries out the necessary operations to prepare for the next measurement with rate r_i^{prep}. Each measurement is accurate with probability p_i, and this probability depends on various factors including the sensor's specification and deployed environment. To save energy, the sensor can be switched on and off through a configurable parameter $x_i \in \{0, 1\}$; if $x_i = 1$ the sensor is active, while if $x_i = 0$ the sensor is switched off. However, switching the sensor on and off consumes an amount of energy given by e_i^{on} and e_i^{off}, respectively.

Figure 2.3 depicts the CTMC model of the i-sensor, adapted from [94]. The model corresponds to a session during which the sensor is either operational ($x_i = 1$) or switched off ($x_i = 0$). If the sensor is switched off, the model transitions to state s_4 with rate r_i^{off} and stays there indefinitely. When the sensor is operational, the model initially moves to starting state s_1 with rate r_i^{on}. The sensor then starts executing and takes measurements with rate r_i. With probability p_i the measurement is accurate and the model transitions to success state s_2; otherwise, it transitions to fail state s_3.

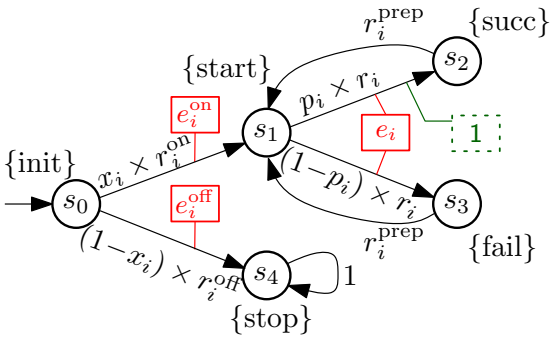


Figure 2.3: CTMC model of the i-th UUV sensor.

Once this event is completed, the model transitions back to state s_1 with rate r_i^{prep} and continues taking measurements for the duration of the session.

The CTMC is augmented with two cost/reward structures, whose non-zero elements are shown in Figure 2.3 in rectangular and dashed rectangular boxes, respectively. The former, "energy" structure associates the energy used to switch the sensor on (i.e., e_i^{on}) and off (i.e., e_i^{off}) and to perform a measurement (i.e., e_i) with the CTMC transitions that model these events. The other cost/reward structure, called "measurement", associates a reward of 1 with the transition corresponding to an accurate measurement.

The model has the following elements: the set of states $S = \{s_0, s_1, s_2, s_3, s_4\}$, the initial state s_0, the set of atomic propositions $AP = \{init, start, fail, succ, stop\}$, and the labelling function $L : L(s_0) = \{init\}, L(s_1) = \{start\}, L(s_2) = \{succ\}, L(s_3) = \{fail\}, L(s_4) = \{stop\}$.

The corresponding transition rate matrix $\mathbf{R}$ and the embedded DTMC $\mathbf{P}^{emb(C)}$ are

$$\mathbf{R} = \begin{pmatrix} 0 & x_i \times r_i^{on} & 0 & 0 & (1-x_i) \times r_i^{\text{off}} \\ 0 & 0 & p_i \times r_i & (1-p_i) \times r_i & 0 \\ 0 & r_i^{prep} & 0 & 0 & 0 \\ 0 & r_i^{prep} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{P}^{emb(C)} = \begin{pmatrix} 0 & \frac{x_i \times r_i^{on}}{x_i \times r_i^{on} + (1-x_i) \times r_i^{\text{off}}} & 0 & 0 & \frac{(1-x_i) \times r_i^{\text{off}}}{x_i \times r_i^{on} + (1-x_i) \times r_i^{\text{off}}} \\ 0 & 0 & p_i & (1-p_i) & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Finally, suppose that the i-the sensor is active, i.e, $x_i = 1$ and $r_i^{on} = 10$. We want to establish the probability of reaching the start state s_1 within 0.2 seconds; thus, $I_0 = [0, 0.2]$. Using (2.6), the probability is equal to

$$\begin{aligned} Pr_s(C(s_0, I_0, s_1)) &= Pr_s(C(s_0)) \cdot \mathbf{P}^{emb(C)}(s_0, s_1) \cdot \left(e^{-E(s_0) \cdot infI_0} - e^{-E(s_0) \cdot supI_0}\right) \\ &= 1 \cdot x_i \cdot (e^0 - e^{-0.2 r_i^{on}}) \\ &= 1 - e^{-0.2 r_i^{on}} \\ &= 0.8646647 \end{aligned}$$

2.1.2 Probabilistic Temporal Logics

Having specified the behaviour of a software system in a Markov model variant (e.g., DTMC or CTMC), the interest now is on quantifying several QoS properties for this system. A property represents a quality aspect of a system, typically quantifiable using specific metrics through well-defined processes; for example, reliability, response time and cost. **Probabilistic temporal logics** is a set of specification languages used for specifying formally the required QoS properties of a system and for reasoning about system behaviour over time. Since transitions in Markov models are associated with a probabilistic choice, we are typically interested in computing the likelihood of an event occurring, instead of simply determining whether the event holds or not. An example QoS property for the e-commerce system from Figure 2.2 is "the probability of a failure ever occurring must be less than 10%". This section introduces Probabilistic Computation Tree Logic (PCTL) [22, 111] and Continuous Stochastic Logic (CSL) [11, 12], the temporal logic variants used to formalise the properties of DTMCs and CTMCs, respectively.

2.1.2.1 Probabilistic Computation Tree Logic

Probabilistic Computation Tree Logic (PCTL) [22, 111] is a branching-time temporal logic for describing properties of DTMCs. To this end, PCTL extends the non-probabilistic Computation Tree Logic [13, 182] with a probabilistic operator P. In this thesis, we use the cost-reward augmented PCTL variant with the syntax from [148], as detailed below.

Definition 2.7. *The syntax of Probabilistic Computation Tree Logic (PCTL) is given by the following grammar:*

$$\Phi ::= true \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid P_{\bowtie p}\,[\phi]$$

$$\phi ::= X\Phi \mid \Phi \bigcup^{\leq k} \Phi$$

and the cost/reward augmented PCTL state formulae are defined by the grammar:

$$R_{\bowtie r}[C^{\leq k}] \mid R_{\bowtie r}[I^{=k}] \mid R_{\bowtie r}[F\ \Phi]$$

where:

- $a \in AP$ *is an atomic proposition with* AP *being a set of atomic propositions;*
- $\bowtie \in \{<, \leq, \geq, >\}$ *is a relational operator;*
- $k \in \mathbb{N} \cup \{\infty\}$*;*
- $p \in [0, 1]$ *is a probability bound (or threshold);*
- $r \in R_{\geq 0}$ *is a reward bound.*

In order to analyse the properties of a DTMC model, PCTL formulae specify conditions over the states of the model. In the definition above, state formulae Φ and path formulae ϕ are evaluated over model states and paths, respectively. Note that path formulae can only occur within the scope of the probabilistic operator $P_{\bowtie p}\,[\cdot]$. This operator defines upper or lower bounds on the probability of system evolution. For instance, a state s satisfies a formula $P_{\bowtie p}[\phi]$ if the probability of the future system evolution meets the bound $\bowtie p$. For a path π, the "next" formula $X\Phi$ holds if Φ is satisfied in the next state. The "bounded until" formula $\Phi_1 \bigcup^{\leq k} \Phi_2$ holds if before Φ_2 becomes true at time step x, where $x \leq k$, Φ_1 is satisfied continuously at time steps

$0, 1, \ldots, x-1$. If $k = \infty$, the formula is termed "unbounded until". Finally, $P_{=?}[\phi]$ can be used to quantify the probability of a path formula ϕ.

Given a state s, the high-level interpretation of the cost/reward operator R is:

- $R_{\bowtie r}[C^{\leq k}]$ holds, if from state s the expected cumulative reward up to time step k meets the bound $\bowtie r$;
- $R_{\bowtie r}[I^{=k}]$ is true if the expected state reward at time step k satisfies $\bowtie r$;
- $R_{\bowtie r}[F\ \Phi]$ holds, if from state s the expected cumulative reward before reaching a state that satisfies Φ meets the bound $\bowtie r$.

As before, $R_{=?}[\cdot]$ can be used to quantify over states and transitions, and to compute the expected value of a reward.

Formally, the semantics of PCTL over DTMCs are defined as follows.

Definition 2.8. *Let $D = (S, \bar{s}, \boldsymbol{P}, L)$ be a labelled DTMC. For any state $s \in S$, $k \in \mathbb{N} \cup \{\infty\}$ and $r \in \mathbb{R}_{\geq 0}$, the satisfaction relation $\models$ is defined inductively by:*

$$\begin{aligned}
s &\models \mathit{true} \quad \text{for all } s \in S \\
s &\models a \;\Leftrightarrow\; a \in L(s) \\
s &\models \neg\Phi \;\Leftrightarrow\; s \not\models \Phi \\
s &\models \Phi_1 \wedge \Phi_2 \;\Leftrightarrow\; s \models \Phi_1 \wedge s \models \Phi_2 \\
s &\models P_{\bowtie p}[\phi] \;\Leftrightarrow\; Pr(s \models \phi) \bowtie p
\end{aligned}$$

where $Pr(s \models \phi) = Pr_s(\pi \in Path^D(s) | \pi \models \phi)$ is the probability that a path starting from s satisfies ϕ.

Moreover, for any path $\pi \in Path^D(s)$

$$\begin{aligned}
\pi &\models P_{\bowtie p}[X\Phi] \;\Leftrightarrow\; \pi(1) \models \Phi \\
\pi &\models P_{\bowtie p}[\Phi_1 \bigcup^{\leq k} \Phi_2] \;\Leftrightarrow\; \exists 0 \leq i \leq k.(\pi(i) \models \Phi_2 \wedge \forall 0 \leq j < i.(\pi(j) \models \Phi_1)) \\
\pi &\models P_{\bowtie p}[\Phi_1 \bigcup \Phi_2] \;\Leftrightarrow\; \exists i \geq 0.(\pi(i) \models \Phi_2 \wedge \forall 0 \leq j < i.(\pi(j) \models \Phi_1))
\end{aligned}$$

Finally, for the cost/reward structures

$$\begin{aligned}
s &\models R_{\bowtie r}[C^{\leq k}] \;\Leftrightarrow\; Exp^D(s, X_{C^{\leq k}}) \bowtie r \\
s &\models R_{\bowtie r}[I^{=k}] \;\Leftrightarrow\; Exp^D(s, X_{I^{=k}}) \bowtie r \\
s &\models R_{\bowtie r}[F^{\Phi}] \;\Leftrightarrow\; Exp^D(s, X_{F^{\Phi}}) \bowtie r
\end{aligned}$$

where $Exp^D(s, X_\Theta)$ gives the expected reward X_Θ over the paths starting at s.

Evaluating a PCTL formula for a DTMC

The algorithm for model checking a PCTL formula takes as inputs a labelled DTMC $D = (S, s_0, \mathbf{P}, L)$ and a PCTL state formula Φ. First, the set of states satisfying Φ is determined. When the question is whether a given state s satisfies Φ, it is sufficient to check if s is in that set. However, if the focus is on quantitative results of the form $P_{\bowtie p}[\phi]$, we need to compute the probability for all states s of the DTMC satisfying formula ϕ, and then compare these values to the bound p.

Model checking DTMCs against PCTL formulae involves the combination of graph traversal algorithms and analytical solution approaches. The former is mainly used for reachability analysis, for example, to examine whether it is possible from the initial state to reach a failure state. Computing the likelihood of an event occurring is carried out by analytical techniques. In particular, formula $P_{\bowtie p}[X\Phi]$ requires one matrix-vector multiplication, while the result of formulae specifying bounded until probabilities $\Phi_1 \bigcup^{\leq k} \Phi_2$, instantaneous rewards $R_{\bowtie r}[I^{=k}]$, and cumulative rewards $R_{\bowtie r}[C^{\leq k}]$ can be estimated using k matrix-vector multiplications. Finally, computing unbounded until probabilities $P_{\bowtie p}[\Phi_1 \bigcup \Phi_2]$ and cumulative rewards $R_{\bowtie r}[F\ \Phi]$ reduces to solving a system of linear equations. We refer the interested reader to [148, 182] for a complete description of the technical details.

Example 2.3. Consider again the e-commerce system with the DTMC model from Figure 2.2. Table 2.1 shows a set of example QoS requirements, including an informal description and their formalisation in PCTL.

Table 2.1: QoS requirements for the train booking system

ID	Informal description	PCTL
R1	(*Workflow reliability*): "Workflow executions must complete successfully with probability at least 90%"	$P_{\geq 0.9}[Fs = s_4]$
R2	(*Buy probability*): "A customer is expected to purchase at least one ticket within the first 7 time steps with probability at least 85%"	$P_{\geq 0.85}[F^{\leq 7} s = s_2]$
R3	(*No shipping*): "At least 75% of the purchased tickets are printed by customers"	$P_{\geq 0.75}[\neg s = s_3 \bigcup s = s_4]$
R4	(*Search cost*): "The expected cost incurred because of searching for tickets during the first 10 time steps must be less than 10 cents"	$R^{\text{"search"}}_{\leq 0.10}[C^{\leq 10}]$

2.1.2.2 Continuous Stochastic Logic

Continuous Stochastic Logic (CSL) [11, 12] is the counterpart of PCTL for specifying properties for CTMC models. CSL extends the non-probabilistic Computation Tree Logic with a probabilistic operator P and a steady-state operator S. We define below the syntax of the cost-reward augmented CSL variant adopted in this thesis [145].

Definition 2.9. *The syntax of Continuous Stochastic Logic (CSL) is given by the following grammar:*

$$\Phi ::= true \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid P_{\bowtie p}\,[\phi] \mid S_{\bowtie p}[\Phi]$$

$$\phi ::= X\Phi \mid \Phi \bigcup^{I} \Phi$$

and the cost/reward augmented CSL state formulae are defined by the grammar:

$$R_{\bowtie r}[C^{\leq T}] \mid R_{\bowtie r}[I^{=T}] \mid R_{\bowtie r}[F\ \Phi] \mid R_{\bowtie r}[S]$$

where:

- $a \in AP$ *is an atomic proposition with* AP *being a set of atomic propositions;*
- $\bowtie \in \{<, \leq, \geq, >\}$ *is a relational operator;*
- $I \subseteq \mathbb{R}_{\geq 0}$ *and* $T \in \mathbb{R}_{\geq 0}$ *are a time interval and a time instant, respectively;*
- $p \in [0, 1]$ *is a probability bound (or threshold);*
- $r \in R_{\geq 0}$ *is a reward bound.*

CSL formulae are analysed over the states of a CTMC model. CSL path formulae are interpreted as for PCTL, except for the interval $I \in \mathbb{R}_{\geq 0}$ parameter of the "until" operator U. For completeness, we describe the semantics of CSL formulae. The probabilistic operator P specifies upper or lower bounds on the probability of system evolution. For instance, formula $P_{\bowtie p}[\phi]$ is true if the probability of future system evolution satisfying ϕ meets the bound $\bowtie p$. For a path π, the "next" formula $X\Phi$ holds if Φ is satisfied in the next state. The "time-bounded until" formula $P_{\bowtie p}[\Phi_1 \bigcup^{\leq I} \Phi_2]$ holds, if across all paths, at some time instant in the interval I, Φ_2 becomes true and Φ_1 holds continuously before. When $I = [0, \infty]$, we obtain an "unbounded until" formula. The steady-state operator S defines the system behaviour in the long-run. Hence, formula $S_{\bowtie p}[\Phi]$ holds,

if the steady-state probability of the system being in a state satisfying Φ meets the bound $\bowtie p$. Finally, the formula $P_{=?}[\phi]$ establishes the probability of path formula ϕ.

The high-level meaning of the cost/reward operator R assuming a target state s is:

- $R_{\bowtie r}[C^{\leq T}]$ is true if the expected cumulative reward up to time T satisfies $\bowtie r$;
- $R_{\bowtie r}[I^{=T}]$ holds if the expected value of the reward at time instant T satisfies $\bowtie r$;
- $R_{\bowtie r}[F\Phi]$ holds if the expected cumulative reward before reaching a state satisfying Φ meets bound $\bowtie r$;
- $R_{\bowtie r}[S]$ is true if the average expected reward in the long-run satisfies $\bowtie r$.

Similarly to PCTL, the formula $R_{=?}[\cdot]$ can be used to calculate the expected value of a reward. The semantics of cost-reward augmented CSL over CTMCs are as follows.

Definition 2.10. *Let $C = (S, \bar{s}, \boldsymbol{R}, L)$ be a labelled CTMC. For any state $s \in S$, time interval $I \in \mathbb{R}_{\geq 0}$, time instant $T \in \mathbb{R}_{\geq 0}$, and reward $r \in \mathbb{R}_{\geq 0}$, the satisfaction relation $\models$ is defined inductively by:*

$$
\begin{aligned}
s &\models \mathit{true} \quad \text{for all } s \in S \\
s &\models a \;\Leftrightarrow\; a \in L(s) \\
s &\models \neg\Phi \;\Leftrightarrow\; s \not\models \Phi \\
s &\models \Phi_1 \wedge \Phi_2 \;\Leftrightarrow\; s \models \Phi_1 \wedge s \models \Phi_2 \\
s &\models P_{\bowtie p}[\phi] \;\Leftrightarrow\; Pr(s \models \phi) \bowtie p
\end{aligned}
$$

where $Pr(s \models \phi) = Pr_s(\pi \in Path^C(s) | \pi \models \phi)$ is the probability that a path originating in s satisfies ϕ.

Moreover, for any path $\pi \in Path^C(s)$

$$
\begin{aligned}
\pi &\models P_{\bowtie p}[X\Phi] \;\Leftrightarrow\; \pi(1) \models \Phi \\
\pi &\models P_{\bowtie p}[\Phi_1 \textstyle\bigcup^I \Phi_2] \;\Leftrightarrow\; \exists t \in I.(\pi@t \models \Phi_2 \wedge \forall j \in [0,t).(\pi@j \models \Phi_1)
\end{aligned}
$$

Finally, for the cost/reward structures

$$
\begin{aligned}
s &\models R_{\bowtie r}[C^{\leq k}] \;\Leftrightarrow\; Exp^C(s, X_{C^{\leq T}}) \bowtie r \\
s &\models R_{\bowtie r}[I^{=k}] \;\Leftrightarrow\; Exp^C(s, X_{I^{=T}}) \bowtie r \\
s &\models R_{\bowtie r}[F^{\Phi}] \;\Leftrightarrow\; Exp^C(s, X_{F^{\Phi}}) \bowtie r \\
s &\models R_{\bowtie r}[S] \;\Leftrightarrow\; \lim_{t\to\infty} \tfrac{1}{t} \cdot Exp^C(s, X_{C^{\leq T}}) \bowtie r
\end{aligned}
$$

where $Exp^C(s, X_\Theta)$ denotes the expected reward X_Θ over the paths starting at s.

Evaluating a CSL formula for a CTMC

The model checking algorithm for a CSL formula Φ takes as input a labelled CTMC and outputs the set of states satisfying Φ. The algorithm for non-probabilistic formulae, i.e., "true", "a", $\neg\Phi$, and $\Phi \wedge \Phi$, is similar to PCTL for DTMCs, which proceeds by induction on the parse tree of Φ. Evaluating properties that involve the probabilistic P or reward R operators is achieved through analytical techniques. Untimed properties, i.e., properties that do not express any of the real time aspects of a CTMC, can be evaluated using the embedded DTMC (Def. 2.5). As with PCTL, the "next" formula $P_{\bowtie p}[X\Phi]$ requires one matrix-vector multiplication. Calculating unbounded until probabilities $P_{\bowtie p}[\Phi_1 \bigcup \Phi_2]$, steady-state probabilities $S_{\bowtie p}[\Phi]$, reachability rewards $R_{\bowtie r}[F\Phi]$, and steady-state rewards $R_{\bowtie r}[S]$ is done by solving a system of linear equations. For timed properties, including probabilistic bounded until $P_{\bowtie p}[\Phi_1 \bigcup^I \Phi_2]$, cumulative $R_{\bowtie r}[C^{\leq T}]$ and instantaneous $R_{\bowtie r}[I^{=T}]$ rewards, the problem reduces to calculating the transient probabilities of the CTMC, using efficient iterative numerical methods such as uniformisation and matrix-vector multiplications. A comprehensive analysis of the techniques used to quantify each CSL formula is presented in [148, 182].

Example 2.4. Suppose an unmanned underwater vehicle (UUV) travelling with speed *sp* is equipped with a set of sensors. The behaviour of each sensor is defined by the CTMC model in Figure 2.3. Let "accurate" indicate the state in which any of the sensors performs an accurate measurement. A set of example QoS requirements for the UUV system is given in Table 2.2. For each requirement, we provide an informal description and its formalisation in cost/reward augmented CSL.

Table 2.2: QoS requirements for the UUV system

ID	Informal description	CSL
R1	(*Sensor liveness*): "The probability that during the first second of operation, the sensors make an accurate measurement must be at least 95%"	$P_{\geq 0.95}[true \bigcup^{[0,1]} \text{“}accurate\text{”}]$
R2	(*Sensor accuracy*): "At least 300 measurements of sufficient accuracy must be taken every 100m travelled by the UUV"	$R^{\text{“measurement”}}_{\geq 300}[C^{\leq 100/sp}]$
R3	(*Energy consumption*): "The energy consumption of the sensor must not exceed 400J per 100m travelled by the UUV"	$R^{\text{“energy”}}_{\leq 400}[C^{\leq 100/sp}]$

2.2 Runtime Quantitative Verification

Quantitative verification has been traditionally used during the off-line stages of a software system's lifecycle. In particular, the technique is used at design time to evaluate possible system architectures before proceeding to implementation, while during maintenance it is used to support system modification as a result of QoS requirements violation [145]. On the other hand, self-adaptive software and software-controlled systems are expected to comply throughout their lifetime with strict dependability, performance and other QoS requirements. To achieve this, self-adaptive systems modify their behaviour and internal structure in response to changing environmental conditions, evolving requirements and internal changes [45, 56]. Thus, the current form of the technique cannot manage scenarios encountered by self-adaptive systems in which they are expected to evolve autonomously while providing service.

To illustrate this scenario, assume that the train booking website from Figure 2.2 uses third-party services for its "*buy*" and "*shipping*" operations. These third-party services are offered by external service providers, typically deployed on a cloud infrastructure. Given that the environment is highly dynamic, cloud-deployed services could experience service degradation for various reasons including datacentre overload and network congestion. The result of this situation could lead to violation of QoS requirements, e.g., workflow reliability is lower than 90% (cf. requirement R1 from Table 2.1). The self-adaptive system should be able to detect or anticipate this violation and select from a pool of functionally equivalent services, these services that restore or maintain compliance with QoS requirements.

We describe next the runtime use of quantitative verification (Section 2.2.1). We also present the integration of quantitative verification within the runtime adaptation process and its use to improve the dependability of self-adaptive software systems (Section 2.2.2). Finally, we overview recent advances in runtime quantitative verification focusing on approaches that improve the efficiency of the technique (Section 2.2.3).

2.2.1 Self-Adaptation Through Runtime Quantitative Verification

Self-adaptive systems are typically engineered using feedback control loops [27]. As highlighted by the SEfSAS[1] research community [27, 45], the MAPE closed control loop is a suitable feedback mechanism for realising self-adaptation in software systems [91, 139]. A MAPE-based self-adaptive system, depicted in Figure 2.4, comprises a managed el-

[1]Software Engineering for Self-Adaptive Systems

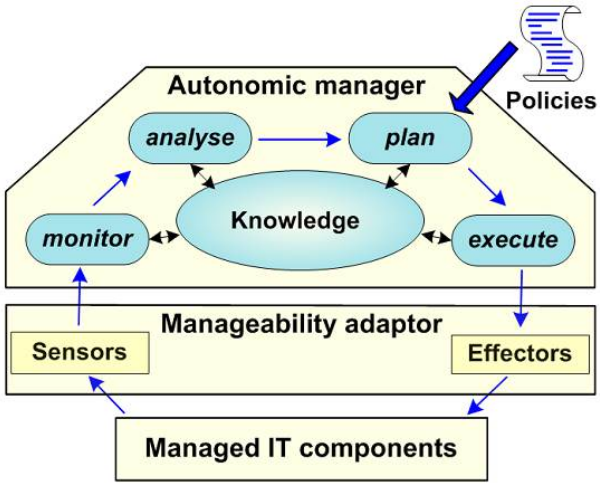


Figure 2.4: High-level architecture of a self-adaptive system implementing the MAPE-K closed control loop [139].

ement and an autonomic manager (or controller). The manager consists of a set of sensors, a set of effectors and the monitor-analyse-plan-execute stages. A knowledge repository, shared across the stages of the MAPE loop, is used to store system requirements, events of interest and updated system models. During *monitoring*, the controller collects information about the managed element and its environment through *sensors*, and updates the knowledge repository. Any issues related to the current system behaviour including violation of system requirements are diagnosed during the *analysis* stage. When a violation occurs, a *planning* stage is responsible for devising a plan to restore compliance with system requirements. Finally, during the *execution* stage the plan is implemented to the managed element through *effectors*.

Runtime quantitative verification (RQV) can drive reconfiguration in self-adaptive systems by supporting the "analyse" and "plan" stages of the MAPE control loop. More specifically, the technique can support adaptation decisions through continual verification of Markov models (cf. Section 2.1.1). RQV was introduced in [38, 68] and further refined by recent research in [33, 76, 77, 130]. Despite being a relatively new approach in the area of self-adaptive systems, RQV has been applied successfully in several application domains. Some illustrative examples include QoS optimisation in service-based systems [33, 34, 77], dynamic reconfiguration of cloud computing infrastructure [38, 130] and adaptive resource management in embedded and robotic systems [31, 65, 94].

In contrast to its off-line counterpart in which transitions are associated with numeric values, RQV operates on parametric Markov models which allow transitions to

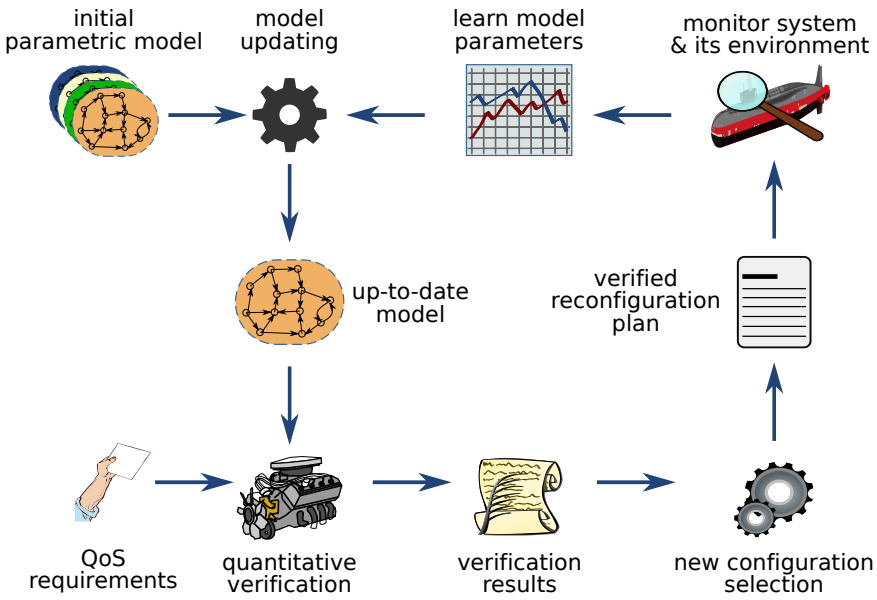


Figure 2.5: Runtime quantitative verification workflow.

be labelled both with numeric values and parameters. Thus, the transition probability matrix $\mathbf{P}$ or transition rate matrix $\mathbf{R}$ may include any of these. A numeric value between a pair of states (source $\rightarrow$ target) is used to specify a known and fixed aspect of the system behaviour; e.g., the failure probability of search operation from the e-commerce DTMC model in Figure 2.2 is 0.01. On the other hand, parameters correspond to aspects of system behaviour that are unknown at design time and/or are subject to change at runtime, e.g., due to environment uncertainty. For instance, user behaviour associated with the buy operation might vary depending on the period of the year (e.g., in Easter and Christmas). The values of these parameters become known only at runtime. Using control theory terminology, we call these parameters *observable*. Another set of parameters, called *configurable*, represents system variability in the form of system configurations selectable for adapting a system. For any given evaluation of all *observable* and *configurable* parameters, a nonparametric model is produced which can then be used to verify QoS system requirements.

Figure 2.5 shows the workflow of quantitative system verification at runtime. The technique involves monitoring the system and its environment continually to establish the current environmental parameters and system state. Relevant changes are diagnosed and quantified using fast on-line learning techniques. This operation enables the selection of a suitable concrete model from a family of parametric system models that correspond to different system scenarios. The selected model is then analysed using

quantitative verification to identify (and in some cases to predict) violations of QoS requirements such as response time, availability and cost. When requirement violations are identified or predicted, the verification results enable the synthesis of a verified reconfiguration plan. This plan consists of adaptation steps whose execution is guaranteed to restore or maintain system compliance with its QoS requirements despite the changes identified in the monitoring stage.

In the following section, we illustrate the application of RQV using an embedded system from the unmanned underwater vehicle (UUV) domain. We developed (as part of this project) the description of the system, and the corresponding Markov model and QoS requirements.

2.2.1.1 Self-Adaptive Unmanned Underwater Vehicle System

UUVs are increasingly used in a wide range of oceanographic and military tasks, including oceanic surveillance (e.g., to monitor pollution levels and ecosystems), undersea mapping, and mine detection. Due to limitations in the environment in which these vehicles operate (e.g., impossibility to maintain UUV-operator communication during missions and high frequency of unexpected changes), UUVs are expected to be self-adaptive [189]. These systems are also safety critical (e.g., when used for mine detection and surveillance of ecosystems that should not be impacted) and/or business critical, since UUVs are often expensive equipment that should not be lost during missions.

The self-adaptive UUV system in our study is deployed to carry out a data gathering mission. The UUV is equipped with $n \geq 1$ on-board sensors that can measure the same attribute of the ocean environment (e.g., water current, salinity or thermocline). The CTMC model of a typical sensor used in this mission has the structure shown in Figure 2.3. We designed the sensor model based on information for real-world underwater sensors[2]. We use the subscript "i" to denote model parameters and transition rates associated with the i-th sensor; e.g., r_i and e_i correspond to the measurement rate and energy consumption of the i-th sensor, respectively. The probability p_i that a measurement is accurate depends on the configurable UUV speed $sp \in [0, 5m/s]$ and a sensor-specific accuracy factor $\alpha_i \in (0, 0.15)$, and is given by $p_i = 1 - \alpha_i sp$.

In a dynamic environment, the UUV is required to adapt to changes in the measurement rates $r_1, r_2, ..., r_n$ of its n sensors and to sensor failures by continually adjusting:

- the UUV speed sp

[2]for example, `http://www.ashtead-technology.com/rental-equipment/rdi-300khz-navigator`

- the sensor configuration $x_1, x_2, ..., x_n$ (where $x_1 = 1$ if the i-th sensor is on and $x_i = 0$ otherwise)

so that the UUV complies with the QoS requirements R1-R3 in Table 2.2.

If requirements R1-R3 are satisfied by multiple configurations, the UUV must use a configuration that maximises the utility function

$$utility(x_1, x_2, ..., x_n, sp) = w_1 sp + w_2/E \tag{2.7}$$

where E is the energy used by the sensors per 100m travelled by the UUV, and $w_1, w_2 > 0$ are weighting coefficients that reflect the relative importance of the UUV speed sp and energy consumption E.

Example 2.5. Given the UUV system description above, the UUV uses quantitative verification at runtime to achieve compliance with requirements R1–R3 and maximise utility (2.7) as follows. An initial parametric model M of the entire n-sensor system is obtained through the parallel composition of CTMC models of the n sensors, i.e., $M = M_1||M_2||...||M_n$, where M_i corresponds to the i-th sensor CTMC model. The measurement rates $r_1, r_2, ..., r_n$ of the n sensors comprise the set of observable parameters, while the set of configurable parameters contains the sensor configuration $x_1, x_2, ..., x_n$ and the UUV speed sp. Recall that, QoS requirements of the UUV system are formalised in the appropriate probabilistic temporal logic as shown in the last column in Table 2.2.

While the UUV system is running, the n sensors are monitored continually to establish the actual measurement rates $r_1, r_2, ..., r_n$. This information is then used to update the model M so that it represents the current behaviour of the sensors. Next, the updated model and the formalised QoS requirements are used to carry out quantitative verification. The outcome of this process is a set of verification results. This set contains an evaluation for each QoS property associated with a system requirement for a range of possible instantiations of the configurable parameters. As an example, Figure 2.6 shows the verification results for a UUV with $n = 2$ sensors with current measurement rates $r_1 = 5s^{-1}$ and $r_2 = 9s^{-1}$. These results establish the probability of

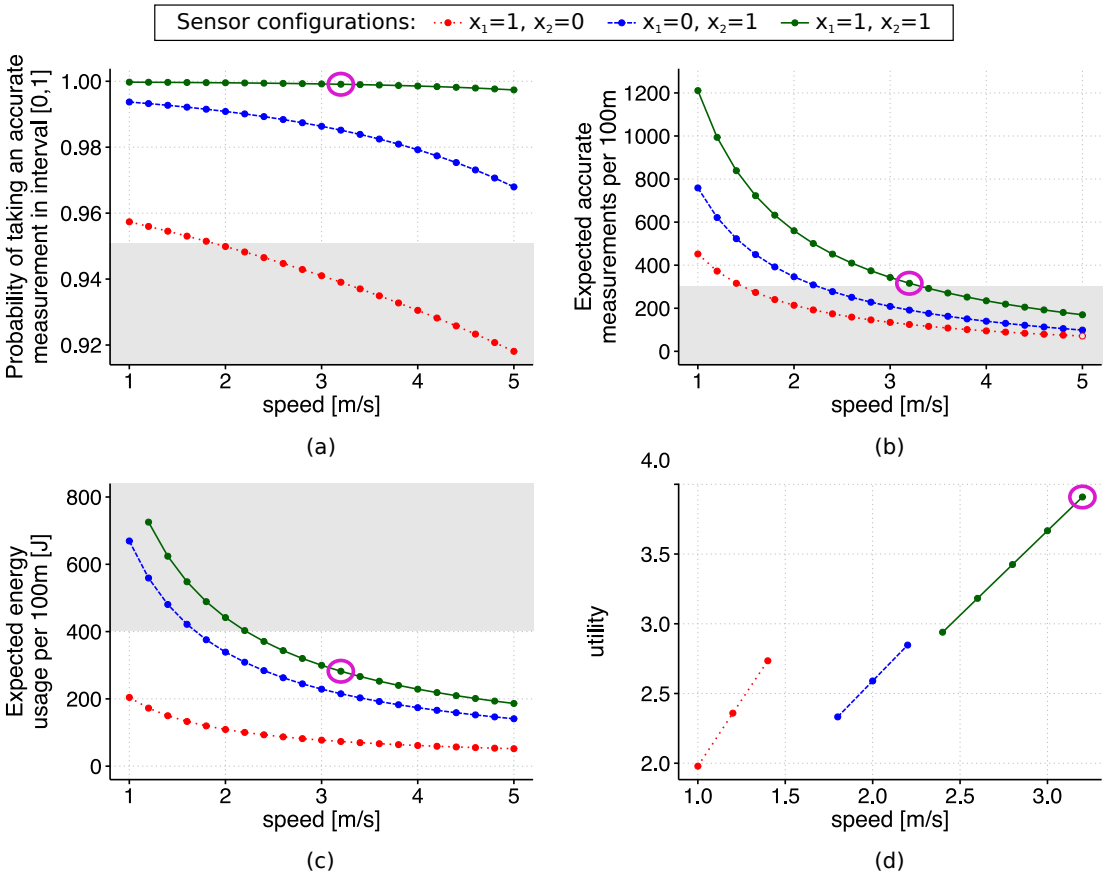


Figure 2.6: Verification results for (a) requirement **R1**, (b) requirement **R2** and (c) requirement **R3** from Table 2.2, and (d) *utility* of the valid configurations of a two-sensor UUV. The configuration associated with the circled results in (a) – (d) is used to reconfigure the system.

taking an accurate measurement in interval [0,1], and the expected number of accurate measurements and the sensor energy consumption per 100m travelled by the UUV.

The technique then filters these verification results to select a new configuration and derive a reconfiguration plan. Any configurations that violate requirements R1, R2 or R3 are discarded. The shaded areas from Figure 2.6 correspond to such configurations. Next, the utility of the remaining feasible configurations is computed. Figure 2.6 shows the utility values associated with the feasible configurations, when using $w_1 = 1$ and $w_2 = 200$ in the utility (2.7). The configuration that maximises the system utility is circled in Figure 2.6. This configuration is used to adapt the UUV system.

2.2.2 Early Approaches to Runtime Quantitative Verification

The very first research efforts at applying RQV focused on augmenting existing or legacy software systems with self-adaptive capabilities. The proposed approaches realised variants of the MAPE control loop and were typically developed through integrating existing software tools and components.

Calinescu and Kwiatkowska [38] propose a user-driven framework that generates self-adaptive versions of existing software systems. Their framework comprises three stages: i) generation, which produces a manageability adaptor specifying the autonomic elements of a system; ii) deployment, in which the adaptor is connected to the software system; and iii) exploitation, in which user-defined policies expressing system objectives are put into use. Off-the-shelf tools and computer-assisted techniques are used to facilitate communication and interaction between the framework stages. System behaviour is described using Markov models (Section 2.1.1) while system QoS properties are formalised in probabilistic temporal logics (Section 2.1.2) and quantified using the probabilistic model checker PRISM [149]. The system is monitored at runtime and once a change is detected, through specified control structures, the system adapts to meet its QoS requirements and user-defined policies. The framework has been evaluated in dynamic power management of disk drives and adaptive control of cluster availability showing its effectiveness and potential applicability.

The work by Calinescu et al. [33] introduces QoSMOS, a generic architecture for developing and managing self-adaptive service-based systems. QoSMOS, short for QoS management and optimisation of service-based systems, is a tool-supported framework that covers the entire scope of the MAPE control loop, from extracting formal specifications of QoS requirements to dynamic reconfiguration of a software system and QoS optimisation. The framework is realised by integrating software components and tools developed previously by the authors, i.e., ProProST [104], KAMI [68], PRISM [149], and GPAC [29]. Using ProProST [104], system QoS requirements expressed in natural language are transformed into formal specifications, i.e., probabilistic temporal logic formulae (cf. Section 2.1.2). KAMI [68] is responsible for analysing runtime system data, identifying changes in system behaviour, and updating the Markov models after revising the model parameters affected by these changes. Quantitative analysis of the probabilistic formulae over the updated system models is done using PRISM [149]. The outcome of this analysis is then forwarded to GPAC [29] that selects the optimal system configuration and adapts the system. Extensive experiments performed on a telehealth service-based system confirmed that QoSMOS is a suitable framework to support recon-

figuration in service-based systems. The authors, however, acknowledge that for large systems, i.e., those comprising many services, QoSMOS requires a considerable amount of time or computational power to carry out quantitative verification. Thus, it might fail to reconfigure the system if time or computational resources are limited.

2.2.3 The Quest for Efficient Runtime Quantitative Verification

RQV drives adaptation in software systems by continually analysing Markov models. Notwithstanding its strengths, the computation and memory overheads required to carry out this analysis depends on the size of the model, which in turn depends on the number of states and the transitions between these states. In fact, the size of the model increases exponentially with the size of the system. For instance, a system comprising n processes, each with m states, operating asynchronously in parallel, has a model of m^n states. Each UVV sensor from Example 2.5 consists of 5 states. Thus, the CTMC model of a UUV equipped with n sensors has 5^n states. This is commonly known as the *state explosion problem* [13, 48]. When applied to large and/or complex self-adaptive systems, whose corresponding models are typically huge, RQV has unacceptably high CPU and memory overheads.

Even if the analysis process can be carried out fast enough for a particular model instantiation (i.e., a model corresponding to a specific system configuration), this is not always sufficient. Many real-world self-adaptive systems have very large configuration spaces, due to the numerous alternative architectures and instantiations of the system parameters. Therefore, identifying a set of optimal configurations to adapt the system is also a computation and memory intensive task. This is another major challenge for RQV, as the technique finds difficulties to meet the strict execution time and resource usage constraints required by this class of self-adaptive systems.

To address these challenges, recent research introduced RQV variants that operate with reduced overheads and improved scalability. We present these variants in the following sections. An overview of these approaches is shown in Table 2.3. The 'Category' column denotes the mechanism used to improve RQV, i.e., *compositional*, *incremental* or *parametric*. The columns 'Model', 'Change' and 'System Type' indicate the type of models (DTMC, CTMC or MDP[3]), changes (structure or transition) and systems (monolithic or distributed) supported by these approaches, respectively.

[3] A Markov decision process (MDP) extends DTMCs by modelling probabilistic systems that exhibit nondeterministic behaviour.

Table 2.3: Overview of surveyed approaches and comparison to new approaches proposes in this thesis

Research work	Category			Model			Change		System Type	
	Increm.	Compos	Param.	DTMC	CTMC	MDP/PA	Struct.	Trans.	Monol.	Distr.
Kwiatkowska et al. [154]	✓			✓		✓		✓	✓	
Forejt et al. [82]	✓			✓		✓	✓	✓	✓	
Bianculli et al. [24]	✓			✓				✓	✓	
Meedeniya et al. [158]	✓			✓				✓	✓	
Kwiatkowska et al. [151]		✓				✓		✓		✓
Calinescu et al. [37]	✓	✓				✓		✓		✓
Johnson et al. [130]	✓	✓				✓	✓	✓		✓
Daws [54]			✓	✓				✓	✓	
Hahn et al. [107, 110]			✓	✓				✓	✓	
Filieri et al. [75, 76, 79]			✓	✓				✓	✓	
Hahn et al. [108]			✓	✓				✓	✓	
Our work										
Chapter 3 Soft. Eng.	✓			✓	✓	✓		✓	✓	
Chapter 4 EvoChecker	✓			✓	✓	✓	✓	✓	✓	
Chapter 5 DECIDE	✓	✓		✓	✓	✓	✓	✓		✓

2.2.3.1 Incremental Verification

Changes affecting a self-adaptive software system are typically localised, i.e., the impact of changes is restricted to specific parts of the system [95]. *Incremental verification* avoids unnecessary computation by exploiting verification results from previous runs. To this end, incremental verification-based approaches try to reverify only model elements associated with the affected system parts and to reuse as much as possible the results obtained from previous reverification steps.

An interesting approach to incremental verification is proposed by Kwiatkowska et al. [154]. The key idea is to decompose the underlying graph of an MDP model into Strongly Connected Components (SCCs)[4]. Each SCC is individually evaluated using value iteration, and then SCC-level evaluations are combined to derive system-level verification results. When changes in transition probabilities occur, the approach uses the topological ordering of SCCs and through a search algorithm identifies the set of SCCs directly and indirectly affected by the changes. Next, the approach reverifies only the affected SCCs and reuses the verification results corresponding to unaffected SSCs. Furthermore, SCCs-based value iteration is ideal for parallelisation because at any step an SCC can be processed independently from other SCCs (certain topological criteria apply). Experimental results on a set of case studies showed significant reduction in computation time. This work, however, can only manage changes in transition probabilities of a Markov model whose structure of the model must remain unaltered.

In [82], Forejt et al. extend the incremental verification technique presented in [154] to both model construction and quantitative verification. Model construction concerns the development of an MDP model from a high-level modelling language such as the PRISM language [147]. Incremental model construction pertains to finding the set of states to be rebuilt after a change, which reduces the set of high-level modelling commands to be evaluated. For quantitative verification, [82] decomposes the system model into SCCs, as in [154]. However, [82] uses policy iteration to establish the correctness of a formula, while [154] applies value iteration. The advantage of using policy iteration incrementally is the capability to reuse policies between verification runs and to specify the adversary (i.e., resolution of MDP nondeterminism) with which the computation starts. This could reduce the number of required iterations and lead to faster convergence. Intuitively, a good initial adversary is the optimal adversary from the previous execution. Unlike [154], the approach supports small changes in the structure of the analysed model. Experiments performed both for model construction and quantitative

[4]A Strongly Connected Component is a set of states in which there is a path between any two states, and which is maximal, i.e., there is no superset that is also strongly connected.

verification showed promising results. For additional information, see [83].

Bianculli et al. [24] introduce a framework for syntax-driven incremental verification. In their proposal, the specification of the structure of a software system conforms to operator precedence grammars. These grammars natively support incremental parsing, which enables the synthesis of incremental verification procedures. When changes occur, the approach employs incremental algorithms for traversing and evaluating the part of the syntax tree affected by the changes. Encouraging results have been obtained after applying the approach to the quantitative verification of reliability properties, but other types of important QoS properties are not supported [23].

The Δ evaluation approach introduced in [158] supports incremental verification of reliability requirements in component-based systems. The behaviour of the analysed system is modelled as a DTMC, where each state of the DTMC corresponds to a system component as suggested in [46]. When a component is in control of the execution, it can either transfer control to another component (based on a workflow), or transition to an absorbing state denoting a failed or successful execution. When a single change occurs in the model, using matrix operations, it is possible to analyse the impact of the change and re-evaluate the reliability of the system without running a complete re-evaluation. The approach can cope with scenarios that require the analysis of reliability requirements of systems affected by a single component change at a time.

An approach similar to Δ evaluation from the domain of model-driven engineering is delta modelling [186]. The conceptual idea of delta modelling is to model system variability by explicitly defining the differences between system variants as deltas. Given a core system and a set of deltas that specify modifications to the core system, new system variants can be derived by applying these deltas to the core system. Delta modelling can also be used to capture the evolution of software systems over time [106].

In contrast to these incremental verification approaches, the complementary RQV variants we introduce in Chapters 3–5 are applicable to both discrete- and continuous-time Markov models (cf. Table 2.3). Furthermore, our variants avoid some of the limiting assumptions of existing approaches, i.e., the need to partition the verified model into much smaller SCCs. Finally, two of the techniques we developed (presented in Chapters 4 and 5) can also cope with structural changes in the verified models.

2.2.3.2 Compositional Verification

Using quantitative verification to analyse QoS requirement compliance of large, complex software systems during runtime is challenging. These systems are typically het-

erogeneous and distributed, characterised by their size and complexity, and comprise independent software components that are subject to continuous change [191]. The monolithic system-level model is derived after the parallel composition of smaller models describing the behaviour of the constituent components. The authors in [37] report that a simple three-tier cloud-deployed software service, composed by four server instances, two web-application instances and two database instances, has a system-level model with $176E+12$ states. Thus, using standard RQV to reconfigure these large-scale systems is intractable due to the huge size of system models.

Compositional verification exploits the component-based structure of large, complex systems and establishes system-level QoS properties from the properties of their components. Techniques within this group perform verification tasks component-wise and infer global system properties based on *assumptions* regarding the acceptable behaviour of system components. A widely used technique is *assume-guarantee reasoning* [171] which specifies that the parallel composition of two models $M_1||M_2$ satisfies a system property G if the following premises hold independently: i) M_2 satisfies G when the component modelled by M_2 satisfies an assumption A; and ii) M_1 satisfies assumption A under all circumstances. This relationship can be formally expressed using Hoare's triple notation [125]:

$$\frac{\langle true\rangle M_1\langle A\rangle \quad \langle A\rangle M_2\langle G\rangle}{\langle true\rangle M_1 \parallel M_2\langle G\rangle} \tag{2.8}$$

Kwiatkowska et al. [151] introduced a probabilistic variant of assume-guarantee reasoning. Probabilistic assume-guarantee is applicable to probabilistic automata, a class of nondeterministic models that generalise MDPs [187]. The technique establishes probabilistic safety properties of the form (2.9). Thus, if model M_1 satisfies property A under any circumstances with probability at least p_1 and M_2 satisfies property G with probability at least p_2 under assumption A, then the system composed by M_1 and M_2 satisfies property G with probability at least p_2. A comprehensive experimental evaluation involving large systems showed that compositional verification achieved execution time and memory consumption several orders of magnitude lower than its non-compositional counterpart. These results enable the application of RQV to much larger systems than previously possible. It should be noted, through, that manual effort is required to produce appropriate assumptions. Recent efforts [72, 73] investigate the use of learning-based techniques for the automatic generation of assumptions.

$$\frac{\langle true\rangle M_1\langle A\rangle_{\geq p_1} \quad \langle A\rangle_{\geq p_1} M_2\langle G\rangle_{\geq p_2}}{\langle true\rangle M_1 \parallel M_2\langle G\rangle_{\geq p_2}} \tag{2.9}$$

A hybrid technique that combines incremental verification with probabilistic assume-guarantee reasoning is proposed by Calinescu et al. [37]. Given the architecture of a component-based system, the technique specifies dependencies between its components using a dependency tree and also associates component models with probabilistic safety properties. When system components are affected by changes, the technique generates the minimal sequence of reverification steps that need to be carried out to re-establish the safety properties. This minimal sequence of steps is derived after performing a depth-first traversal of the dependency tree. Each verification step is carried out using the results of some or all of the previous steps as assumptions. Experiments performed on a dynamically changing cloud computing infrastructure showed promising results.

Johnson et al. [130], building on the work presented in [37], introduce an Incremental Verification Strategy (INVEST) framework for the efficient reverification of component-based software systems after changes such as additions, removals and modifications. INVEST comprises three layers: i) a generic incremental verification engine that identifies the minimal sequence of components requiring reverification after a change; ii) an assume-guarantee model checker that performs the reverification based on the sequence received by the engine; and iii) a domain-specific adaptor that connects the framework to component-based systems and performs incremental verification of their probabilistic safety properties. A prototype INVEST tool was implemented and evaluated in the context of a cloud-deployed software system. The experimental evaluation showed that INVEST can establish probabilistic safety properties in 10-60% of the time taken by other probabilistic assume-guarantee reasoning techniques for a wide range of scenarios.

Despite the potential of the approaches presented in this section, they require significant expertise to generate suitable assumptions, can handle only minor changes in the structure of the verified models (e.g., individual states being added to or removed from the models [130]) and are only applicable to (discrete-time) probabilistic automata. In contrast, the RQV variants introduced in this thesis do not suffer from these limitations. They are model agnostic and thus can be applied to several types of Markov models (cf. Table 2.3). Also, the variants presented in Chapters 4 and 5 can cope with more significant changes in the structure of the verified models (e.g., different system or component architectures).

2.2.3.3 Parametric Verification

Developing self-adaptive systems typically entails considering uncertainty in system operation due to incomplete system specification at design time and changes occurring

in the surrounding environment and the system itself at runtime. *Parametric Markov models* provide the means of specifying these uncertain aspects for the system under consideration. In these models, transition probabilities are not fixed, but are associated with parameters whose values become known only at runtime and might change during the system operation. Parametric model checking [54] is an approach comprising a design-time step and a runtime step, and enables to reason about the satisfaction of QoS requirements with low runtime overheads. At design time, through a computationally expensive pre-computation step, QoS requirements are translated into algebraic expressions. At runtime, once the system is within a concrete environment, these algebraic expressions are evaluated by replacing the unknown parameters with the actual values obtained through system monitoring. This runtime evaluation step takes a fraction of the time required to carry out quantitative verification on the actual system model. The approaches described in the following paragraph focus on the design-time pre-computation step, and more specifically, on deriving the algebraic expressions.

In his pioneering work on parametric model checking, Daws [54] introduces a new language-theoretic approach to symbolic probabilistic model checking of reachability properties over DTMCs. The approach initially converts a DTMC into a finite state automaton in which transition probabilities are modelled as letters of an alphabet. This step is followed by the synthesis of a regular expression that defines the language recognised by the automaton using state elimination algorithms. Subsequently, the derived regular expression is subject to a recursive evaluation that yields a rational algebraic expression for the property to evaluate. Despite its originality and its usefulness in reachability properties, Daw's approach does not support neither the full PCTL nor reward properties. Another limitation of the approach is that the length of the regular expression is affected heavily by the number of model states n, yielding in the worst-case scenario an expression of length $n^{\Theta(logn)}$.

In [107, 110], the authors draw upon the work presented by Daws [54] and present an effective approach that intertwines state elimination with early evaluation of the rational function. In each iteration of the approach, a state elimination step is followed by on-the-fly simplification of the rational function taking advantage of cancellations, symmetries and simplifications of arithmetic expressions. Compared to [54], the algorithm requires n^3 operations in most cases showing significant improvements in incurred overheads. In the worst-case scenario, however, the length of the rational function is still $n^{\Theta(logn)}$. This can occur if no rational function can be simplified during the entire process, a rather uncommon scenario according to the findings [107, 110]. The approach is the core of the model checker PARAM [109] and has been recently implemented in PRISM [149]

and PROPhESY [58].

The WorkingMom framework [76] follows the same principles as [54, 107, 110]. Given a parametric DTMC model of the system and a set of reliability-related QoS requirements, this technique generates a set of algebraic expressions. The computation time depends on the size of the DTMC model, the number of parametric states and the number of outgoing transitions from these states. Extensive experiments reported in [76] for the probabilistic model checkers PRISM [149] and MRMC [133] showed that the time taken by the runtime step of the approach is several orders of magnitude lower than both probabilistic model checkers. An extension of the approach supporting the derivation of algebraic formulae for DTMCs augmented with reward structures is presented in [75]. For an extended version of the works in [76] and [75], see [79].

The work by Hahn et al. [108] takes a different perspective and considers the problem of parameter synthesis of PCTL formulae for parametric models. Instead of generating a rational function that represents a reachability requirement, the approach synthesises the set of parameter values for which the reachability requirement holds. At design time, applying recursively state space exploration techniques, the parameters space is partitioned into hyper-rectangles, i.e., regions in the dimension of the model parameters that represent families of models. Each of these regions provides globally the same output, that is, the requirement holds (or not) for all the concrete models resulting from instantiations of the parameters with values in this region. Note that the approach allows a limited state space area to remain unknown; evaluation in this area is very complex and is left undecided. When the system undergoes changes at runtime, it is sufficient to access these hyper-rectangles and instantaneously assess whether the requirement is still satisfied or not. A preliminary implementation of the approach has been developed as part of PARAM [109].

The approaches presented in this section achieve significant improvements in runtime quantitative verification both in terms of computation time and memory consumption. The computationally expensive model exploration is carried out only once at design time, while runtime complexity reduces to simply evaluating a set of algebraic expressions in [54, 75, 76, 107, 110] or quickly accessing a lookup table in [108]. Enhancing further these approaches to deal with a larger number of parametric transitions with respect to the total number of system transitions as well as structural model changes are threads of current research. However, these approaches are only applicable to discrete-time models, and cannot manage structural changes in the analysed model. As shown in Table 2.3, our RQV variants introduced in the following chapters address these limitations of parametric verification.

Chapter 3

Efficient RQV Using Conventional Software Engineering Techniques

Runtime quantitative verification has been advocated by recent research as a suitable technique to support adaptation in software systems [32, 38, 68]. This is mainly because of the capabilities of the technique to deal with environment uncertainty and unexpected changes to requirements or the system itself. The technique has been successfully applied in various application domains including QoS optimisation in service-based systems [33, 68], and dynamic resource management of cloud infrastructure [37, 130].

Despite its capabilities, RQV suffers from the *state-explosion problem* [48], which limits the size of models that it can manage at runtime without unacceptable overheads. The approaches discussed in Section 2.2.3 are a first step towards reducing these overheads and extending the use of the technique to larger models. Each of these approaches achieves reductions in execution time and/or resources required to perform an RQV step, i.e., to carry out the analysis, to interpret the results, and, if needed, to assemble and execute a reconfiguration plan. Their applicability, however, is limited to certain self-adaptation scenarios and to specific types of stochastic models and properties (see Table 2.3).

To improve RQV efficiency further, we need to consider how the behaviour of a software system affects the use of the technique at runtime, and, certainly, how RQV carries out the analysis of an RQV step (i.e., an adaptation). To illustrate these concepts, we use the UUV system from Section 2.2.1.1 which is required to adapt to changes in measurement rates of its on-board sensors by adjusting its speed and the configuration of

sensors. An RQV step is performed either at frequent intervals (e.g., every 5 seconds) or when the sensors undergo changes in their measurement rates (e.g., experience service degradation). When any of these conditions holds, the analysis is executed from the very beginning, irrespective of the extent of the changes. If these changes are minimal, i.e., little difference exists between the current and previously estimated system behaviour, it is possible that similar analysis results would have been obtained in the recent past. Similarly, if the changes affect only a subset of the UUV sensors, i.e., they are localised [95], we could possibly reuse some of the results already available.

In this chapter, taking into consideration these observations, we introduce a set of complementary techniques to advance the state-of-the-art in RQV. These techniques are extensively used in other areas of software engineering; see Section 3.4 for a discussion of related work. To the best of our knowledge, however, they have not been applied to improve RQV efficiency previously.

First, we consider the *caching* of recent verification results. Since changes in real-world systems are often (though by no means always) localised, there is a possibility that verification results from recent RQV steps could be reused if retained for some time. Similar to other applications of caching, the aim is to reduce RQV *response time* (i.e., the time required to perform an RQV step) and CPU usage at the expense of using additional memory.

Second, we augment RQV with *limited lookahead*, which involves using spare CPU cycles to pre-verify stochastic models deemed likely to arise in the future. Since some RQV steps may require the verification of models that were already pre-verified, the technique has the potential to reduce RQV response time at the expense of increased use of CPU and memory.

Finally, we combine RQV with *nearly-optimal reconfiguration*, a technique that terminates an RQV step as soon as (i) a system configuration that satisfies QoS requirements is found; and (ii) a stopping criterion is met, e.g., the selected configuration has a similar utility to the best "utility" encountered over a pre-defined time interval.

The main contribution of this chapter is the integration of RQV with caching, limited lookahead and nearly-optimal reconfiguration, and combinations thereof. We introduce these techniques in Section 3.1. Next, in Section 3.2, we present the extension of the open-source platform MOOS-IvP for the development of autonomous systems with RQV capabilities and also describe the implementation of the techniques within this environment. We analyse our findings in Section 3.3. Finally, in Sections 3.4 and 3.5, we conclude the chapter with a discussion of related work, and with a brief summary of our contributions in this chapter, respectively.

3.1 Techniques for Efficient RQV

3.1.1 Caching

Caching is a fundamental technique for performance optimisation in modern computing systems. A cache is a small memory area, among the multiple storage components used within these systems. Compared to other auxiliary storage (e.g., hard disk), cache is much faster, and thus it shortens data access times, reduces latency and improves input/output. Cache, however, is more expensive than auxiliary storage and therefore its size is typically a fraction of the size of auxiliary storage [60].

The general idea behind caching is based on *locality of reference*, i.e., to store data that might become useful in the near future in a cache memory, so that future requests to accessing the same data can be executed faster [190]. When a request to obtain some data arrives, the cache is checked first. If the data exists in cache, a *cache hit* occurs, and the data is retrieved immediately. Otherwise, there is a *cache miss* and the request is forwarded to the auxiliary storage. In the latter case, upon receiving the response, the data is also stored in cache for future reference. When the cache is full, a *replacement policy* selects which data must be evicted so that new data can be cached. A general principle defining a good replacement policy is to discard data that will not be required for the longest time in the future. A commonly used metric for establishing the efficiency of a replacement policy is the frequency with which the requested data exists in cache, called *hit ratio*. Several cache replacement policies exist; e.g., random replacement, least frequently used, least recently used [172].

Locality is a direct relationship between the changes impacting a software system and the principle underlying caching. Thus, using a cache is a natural mechanism to exploit verification results from previous RQV steps. This would potentially reduce RQV response time and computational overhead, and improve its overall performance.

Our use of caching employs a standard cache using a least frequently used (LRU) replacement policy. LRU is among the most common and most efficient replacement policies [159]. When the cache is full, LRU evicts the data used the least in the recent past assuming that it is less likely to be used again in the near future [172]. Each entry in cache is in the key-value form $\langle (modelParams, propID), (result, timestamp) \rangle$, where:

- $modelParams$ is a set of values where each value corresponds to an evaluation of an observable/configurable parameter from the parametric model of the system;
- $propID$ is the index of a probabilistic temporal logic formula corresponding to a QoS property;

Algorithm 1 RQV with LRU-replacement caching

```
1:  LRUCache ← {}

2:  function VERIFY(modelParams, propID)
3:      key ← (modelParams, propID)
4:      if LRUCache.containsKey(key) then
5:          entry ← LRUCache.get(key)
6:          entry.timestamp ← NOW
7:          result ← entry.res
8:      else
9:          M ← GET_UPDATED_MODEL(modelParams)
10:         Φ ← GET_PROPERTY(propID)
11:         result ← QV(M, Φ)
12:         if LRUCache is full then
13:             Evict LRUCache entry using LRU policy
14:         end if
15:         LRUCache.add(key, (result, NOW))
16:     end if
17:     return result
18: end function
```

- *result* represents the result obtained from the quantitative verification of the property Φ with index *propID* over the model M with parameters *modelParams*;

- *timestamp* is the latest time when this entry was used.

Algorithm 1 shows the pseudocode for this cache-enabled version of RQV. Starting with an empty cache (line 1), the technique uses a VERIFY function to store new verification results into the cache. To this end, the technique uses the *modelParams* and *propID* variables to produce an up-to-date model M (line 9) and to derive the property Φ to be evaluated (line 10), respectively. New results are obtained through the use of QV(M,Φ) function (line 11), and added to the cache according to the LRU policy (lines 12–15) for future use (lines 5–7).

Example 3.1. Consider again the UUV example from Section 2.2.1.1. The set of all *modelParams* instantiations contains the possible sensor measurement rates $r_1, r_2, ..., r_n$ (observable parameters), and the sensor configuration $x_1, x_2, ..., x_n$ and the UUV speed *sp* (configurable parameters). Concerning the other cache entry elements, *propID* $\in \{R1, R2, R3\}$, *result* is a real number and *timestamp* is a time-point in milliseconds.

3.1.2 Limited Lookahead

Limited lookahead is a widely used forward searching technique. It has been applied in many research areas including decision making and planning; see Section 3.4 for a discussion of related work. The technique is particularly useful in software systems whose control decisions have idle times (e.g., periodic execution of an RQV step on a UUV) and/or in scenarios with discrete decision sets (e.g., the number of available third-party implementations for operations "buy" and "shipping" of an e-commerce system).

The principle of limited lookahead involves performing a k-step ahead projection of the behaviour of a software system [47]. Given the current state of the system, the technique carries out a forward search (or simulation) and generates a search tree of depth k. This tree corresponds to the set of control decisions the system can make within this prediction horizon. In its most common form, the decision maximising a utility function is selected and applied to the system. Although this procedure is useful when idle times during system operation exist, it is subject to computational abilities.

We modified the technique with the aim to improve RQV efficiency. Our use of lookahead involves taking advantage of spare computational and memory resources to pre-compute the quantitative verification results associated with *system* and *environment* states "close" to the current state. Under the assumption that real-world systems are often (though not always) evolving through small changes [95], this will ensure that verification results required during an RQV step will sometimes be already available. This pre-computation can take advantage of idle CPU times or may be delegated to an external service, e.g., one that is deployed on cloud computing infrastructure. To give a pictorial view of this technique, this is like creating a "protection zone" around a point associated with the current state and environment conditions. The effect is a decrease in the (average) time required to take decisions, at the expense of using additional computational resources and memory.

The *system* and *environment* states correspond to the sets of parameter values required to produce an up-to-date model from the parametric model of the system. A subset of these parameters is *observable* and cannot be modified by the system, while the others are *configurable* and are used to adapt the system.

Concerning the *observable* parameters, to identify states that are "close" to an m-dimensional state, we calculate the distance between states $s = (s_1, s_2, \ldots, s_m)$ and $s' = (s'_1, s'_2, \ldots, s'_m)$ using the normalised Chebyshev distance:

$$L_\infty(s, s') = \max_{1 \leq i \leq m} \frac{|s_i - s'_i|}{s_i^{\max} - s_i^{\min}}, \tag{3.1}$$

Algorithm 2 RQV with limited lookahead

```
1: function LOOKAHEAD(modelParams, propID)
2:     s ← observable parameter values in modelParams
3:     Cfg ← cartesian product of cofigurable parameters

4:     for all c ∈ Cfg do
5:         for all states s' such that L∞(s, s') ≤ ε do
6:             modelParams' ← modelParams ⊕ s' ⊕ c
7:             key ← (modelParams', propID)
8:             if ¬LRUCache.contains(key) then
9:                 M ← GET_UPDATED_MODEL(modelParams')
10:                Φ ← GET_PROPERTY(propID)
11:                result ←QV(M, Φ)
12:                if LRUCache is full then
13:                    Evict LRUCache entry using LRU policy
14:                end if
15:                LRUCache.add(key, (result, NOW))
16:            end if
17:        end for
18:    end for
19: end function
```

where the values $s_i^{\min}$ and $s_i^{\max}$ represent the minimum and maximum values of state parameter s_i, respectively. A pair of states s and s' is "close" if $L_\infty(s, s') \leq \epsilon$ for some $\epsilon > 0$. The choice of ϵ can be subject to the available processing power or memory, or to the idle time between successive RQV steps.

The pseudocode for our lookahead technique is shown in Algorithm 2. The technique uses a caching mechanism to store the analysis results obtained from lookahead search. The LOOKAHEAD function is invoked whenever spare CPU resources are available immediately after the execution of the VERIFY function from Algorithm 1, and with the same parameters as VERIFY. A lookahead search is performed for all configurable parameters and for all states that are ϵ-close to the current state (lines 4-18).

Example 3.2. For the 2-sensor UUV system from Example 2.5, the values of the observable and configurable parameters are of the form $s = (r_1, r_2)$ and $c = (x_1, x_2, sp)$, respectively. Assuming that the current state $s = (5, 9)$ and $\epsilon = 0.01$, Figure 3.1 shows an excerpt of the search tree produced by our limited lookahead technique.

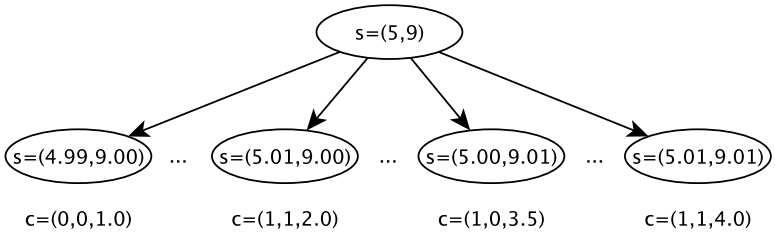


Figure 3.1: Search tree produced by limited lookahead for the 2-sensor UUV system.

3.1.3 Nearly-Optimal Reconfiguration

The task of an RQV-based self-adaptive system can be considered from a different viewpoint as a combinatorial optimisation problem. In this case, the objective is to find a system configuration that satisfies a set of constraints and maximises a utility function. Solving this problem and obtaining an exact solution is typically prohibitively time consuming, if at all possible. In fact, most large real-world optimisation problems (e.g., travelling salesman, bin packing) are intractable, because there is no efficient polynomial-time exact algorithm to solve them.

Given one such problem, a class of algorithms, called approximation algorithms, is capable of finding a near-optimal solution quickly [204]. These techniques do not provide any guarantees regarding the optimality of the obtained solution. Good approximation algorithms, though, can find a feasible solution to a given problem in a reasonable amount of time (at most polynomial). Furthermore, the obtained solution is of "high quality", i.e., close to the optimum.

We devised an approximation algorithm, called *nearly-optimal reconfiguration*, to improve the overall RQV performance. Nearly-optimal reconfiguration is capable of reducing both the response time of RQV steps and their CPU usage. The technique can be used on its own, in conjunction with caching or with caching and lookahead.

The underlying principle of the technique is to select the first valid configuration whose utility is sufficiently close to the best utility (associated with a system configuration) encountered over a long period of time. To this end, the technique continually updates the minimum and maximum utility corresponding to configurations analysed during self-adaptation. We allow an initial learning period $T > 0$ during which the technique is allowed to obtain a good approximation of minimum and maximum utility. After this period has elapsed, the technique accepts valid configurations whose utility satisfies the constraint

$$utility \geq minUtility + \alpha(maxUtility - minUtility) \tag{3.2}$$

Algorithm 3 RQV with nearly-optimal reconfiguration

```
1: minCost ← ∞
2: maxCost ← −∞
3: startTime ← −1

4: function NearlyOptimalReconfiguration(utility)
5:     if startTime = −1 then
6:         startTime ← NOW
7:     end if
8:     minUtility ← (utility < minUtility)?utility : minUtility
9:     maxUtility ← (utility > maxUtility)?utility : maxUtility
10:    if NOW ≥ startTime + T then
11:        return utility ≥ minUtility + α(maxUtility − minUtility)
12:    else
13:        return false
14:    end if
15: end function
```

where $0 < \alpha < 1$ is a parameter that reflects the *lenience* of the technique. As the lenience parameter approaches its upper bound, i.e., $\alpha \approxeq 1$, the acceptance policy becomes stricter.

The pseudocode for our nearly-optimal reconfiguration technique is shown in Algorithm 3. The function NearlyOptimalReconfiguration is invoked when a valid configuration has been identified during an RQV step. If the "learning" period of length T has not been completed or the *utility* does not satisfy the constraint (3.2), the configuration is not accepted, and the function returns false. Otherwise, the configuration associated with that *utility* is accepted and used to reconfigure the system.

Example 3.3. Consider again the UUV system from Example 2.5. The technique relaxes the utility (2.7) so that configurations whose utility is nearly optimal are adopted immediately. Assume a lenience parameter $\alpha = 0.9$ and an initial learning period $T = 10s$, during which we obtained $minUtility = 1.5$, $maxUtility = 5.5$. If no change occurs to the minimum and maximum utility values, any configuration that satisfies requirements R1–R3 and has $utility \geq 5.1$ will be selected to adapt the UUV.

3.2 Implementation

To evaluate the proposed RQV variants, we used the UUV case study from Section 2.2.1. To this end, we implemented a fully-fledged simulator for the self-adaptive UUV system using the open source MOOS-IvP middleware[1] co-developed at MIT and the University of Oxford. MOOS-IvP is a widely used platform for the implementation of autonomous applications on unmanned marine vehicles. The platform is typically deployed on the payload computer of an autonomous vehicle, so as to not interfere with the navigation and control system running on the main vehicle computer [20, 21].

The publish-subscribe architecture of the core MOOS software (Figure 3.2) allows applications to publish messages comprising simple key–value pairs with agreed frequencies. These messages can convey, for instance, information about vehicle components monitored by individual applications or about changes to mission objectives (received from a human operator or a peer unmanned vehicle). Any interested "listener" applications can then act upon these messages, e.g., by adjusting the parameters of the navigation and control system they are responsible for.

In addition, user-implemented MOOS applications can propose *behaviours*, i.e., combinations of boolean logic constraints and parametrised piecewise-linear utility functions. These parameters refer to the navigation and control of the UUV system, including *heading*, *speed* or *depth*. A special component of the platform, the IvP Helm, is responsible for the periodic collection and integration of these proposed behaviours. This component uses Interval Programming (IvP) multi-objective optimisation to "reconcile" the behaviours proposed by all contributing applications, and publishes the optimal solution (i.e., an optimal point in the decision space defined by the constraints and utility functions) as key–value pairs that the other applications can subscribe to receive.

For the UUV system, we developed a Runtime Quantitative Verification MOOS (RQV-MOOS) application (Figure 3.2) that carries out quantitative verification operations using an embedded instance of the PRISM probabilistic model checker [152]. RQV-MOOS operates by:

(i) listening for messages published by the control software for the n sensors, to obtain information about the current rates r_1, r_2, ..., r_n that the sensors operate at;

(ii) carrying out periodic RQV steps, to verify the system compliance with requirements R1–R3, and to select new configurations (i.e., new values for the sensor

[1] `http://oceanai.mit.edu/moos-ivp`

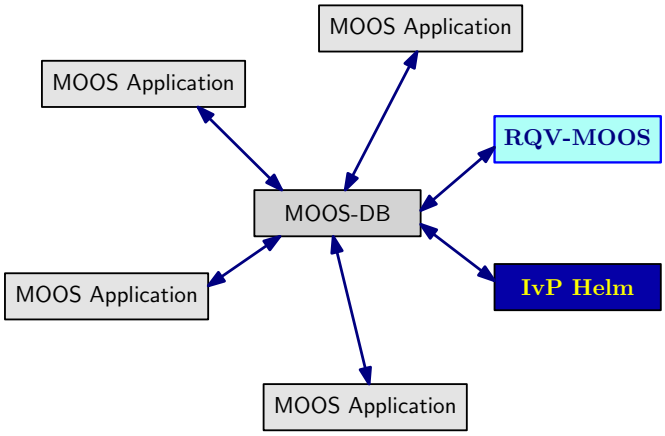


Figure 3.2: MOOS architecture, adapted from [20], including our RQV-MOOS component.

on/off parameters x_1, x_2, $\ldots x_n$ and for the UUV speed sp) that maximise the UUV utility (2.7);

(iii) publishing messages that announce the new sensor configurations, so that the control software for sensor i receives the new x_i, $1 \leq i \leq n$;

(iv) proposing a *behaviour* that recommends to the IvP Helm the new UUV speed sp.

Figure 3.3 shows a screenshot of a 3-sensor instance of our self-adaptive UUV simulator, at a time moment when sensors 1 and 3 are switched on (i.e., $x_1 = x_3 = 1$), sensor 2 is switched off (i.e., $x_2 = 0$), and the UUV speed is $sp = 3.6$m/s. The open-source code for our RQV-MOOS application and UUV simulator, the full experimental results and a video recording of the demo from which we extracted the screenshot in Figure 3.3 are freely available at `http://www-users.cs.york.ac.uk/~simos/SEAMS`.

3.3 Evaluation

3.3.1 Research Questions

The aim of our experimental evaluation was to answer the following research questions:

RQ1 (Validation): Can RQV support dependable self-adaptation in UUVs? This is the first research work applying RQV to the UUV domain. Thus, we wanted to establish the applicability of the technique in this domain and whether it can achieve this with reasonable overheads.

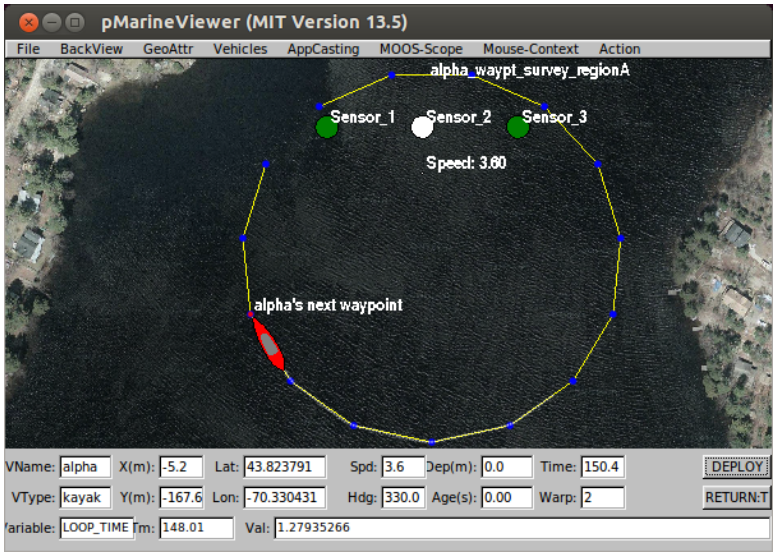


Figure 3.3: Self-adaptive UUV simulator component.

RQ2 (Comparison): How effective are the RQV variants enhanced with the proposed conventional software engineering techniques? With this research question we want to investigate the improvement in the overall RQV performance of the newly introduced RQV variants. To this end, we compare RQV augmented with caching, lookahead, nearly-optimal reconfiguration and combinations thereof against the standard RQV.

RQ3 (Insights): What inferences can be drawn regarding the use of these RQV variants to reconfigure software systems at runtime? We used this research question to establish how well the proposed RQV variants can cope with changes affecting a software system. To this end, we identified the connection between various adaptation scenarios and the effectiveness, benefits and limitations of these RQV variants.

3.3.2 Experimental Setup

To evaluate the effectiveness of RQV augmented with caching, lookahead, nearly-optimal reconfiguration and combinations thereof, we carried out a broad range of experiments using the self-adaptive UUV system variants shown in Table 3.1. In this table, the 'Details' column reports the number of sensors (n) and nominal measurement rate associated with each sensor $r_1, r_2, ..., r_n$. The 'Size' column reports the size of the configuration space that an exhaustive search would need to explore, assuming two-decimal

Table 3.1: Analysed UUV system variants

Variant	Details	Size
UUV_Small	$n = 3, r_1 = 5Hz, r_2 = r_3 = 4Hz$	$2.56E + 11$
UUV_Medium	$n = 4, r_1 = r_4 = 5Hz, r_2 = r_3 = 4Hz$	$2.56E + 14$
UUV_Large	$n = 6, r_1 = r_4 = 5Hz, r_2 = r_3 = r_5 = r_6 = 4Hz$	$1.63E + 20$

precision for the double-valued parameters (i.e., sensors measurement rates and UUV speed). We evaluated these variants on a test suite comprising 8 scenarios, corresponding to the different combinations of the following mission characteristics:

- mission duration of 250, 500, 1000 and 2000 RQV steps (executed every 5s);
- level of sensor rate fluctuations *during periods of normal behaviour*—low (up to 2%) and high (up to 10%).

Beyond normal behaviour, the sensors encounter periods of unexpected behaviour where their rates change dramatically. To simulate this situation, we seeded the scenarios with patterns of sensor failures and/or significant degradation in measurement rates. These scenarios comprise a representative set of events (i.e., sensor rate fluctuation, sensor failure and sensor recovery) that the UUV sensors might encounter during a mission. We focus on sensor rate variation since this is the observable parameter in the CTMC model of the i-th UUV sensor (Figure 2.3).

We carried out separate experiments for each scenario and UUV variant using standard RQV, and RQV augmented with caching, lookahead and caching, nearly-optimal configuration, and nearly-optimal configuration and caching. For each technique and combination of techniques that involved the use of caching, we experimented with cache sizes ranging from 10^4 to 10^6 entries. As shown in Table 3.1, the number of possible configurations per system variant depends on the number of UUV sensors n. For the smallest variant (i.e., when $n = 3$), the *configuration space size* is $2.56E + 11$, while for the largest variant (i.e., when $n = 6$) its size is $1.63E + 20$. Thus, all the cache sizes are orders of magnitude smaller than the amount of memory required to store the verification results associated with all possible system configurations. As far as lookahead is concerned, we set ϵ to 0.002 and 0.02 for the scenarios with low (2%) and high (10%) level sensor rate fluctuations, respectively. Using again two-decimal precision for UUV speed and sensor rates, the upper bound of additional configurations verified after each RQV step is $40 \times 2^n \times 30^n$. The pre-computation step is a best-effort operation, i.e.,

lookahead evaluates the maximum number of configurations within the available time. Finally, for nearly-optimal reconfiguration we allowed an initial learning period $T = 10s$ and fixed its *lenience* parameter value α to 0.9. We ran the experiments on an Intel Core i7-3770 3.40GHZ computer with 8GB of RAM, running Ubuntu 12.04 64-bit.

3.3.3 Results and Discussion

RQ1 (Validation). We start the presentation of our evaluation with a series of results associated with a 3-sensor self-adaptive UUV. These sensors operate with the rates of the UUV_Small variant from Table 3.1 and are experiencing the pattern of variation in measurement rates depicted in Figure 3.4. Active sensors are associated with shaded areas while areas not shaded indicate switched off sensors. When a sensor suffers from service degradation, the system checks the sensor at frequent intervals to assess if it has recovered; these checks are indicated by thin shaded areas. For instance, during mission period $50 - 200s$ the measurement rate of sensor 3 drops to 50% of its nominal rate and the system periodically probes the sensor to establish its current state. The adaptation decisions taken by the system for this scenario are explained below. The entries (A) – (L) refer to the labels in Figure 3.4.

(A) The rate of sensor 3 decreases significantly, and the UUV switches it off and starts using sensor 2.

(B) As sensor 2 also experiences a decrease in rate, the UUV continues with only sensor 1, but needs to decrease its speed considerably in order to obtain sufficient measurements per every 100 metres travelled (cf. requirement R2).

(C-D) Sensor 1 operates with low rate and is switched off; the UUV starts using sensor 3 and reduces its speed.

(E) Sensor 3 recovers and the UUV starts using it along with sensor 1; the speed is increased accordingly.

(F) Sensors that are switched off due to poor performance are periodically tested to find out whether they recovered. Since they may not have recovered, none of the other sensor UUV parameters is modified during these tests.

(G) Sensor 1 operates with decreased rate, so the slightly lower-rate sensor 2 takes over alongside sensor 3, with a suitable lowering of the UUV speed.

(H) Sensor recovery is not always detected immediately.

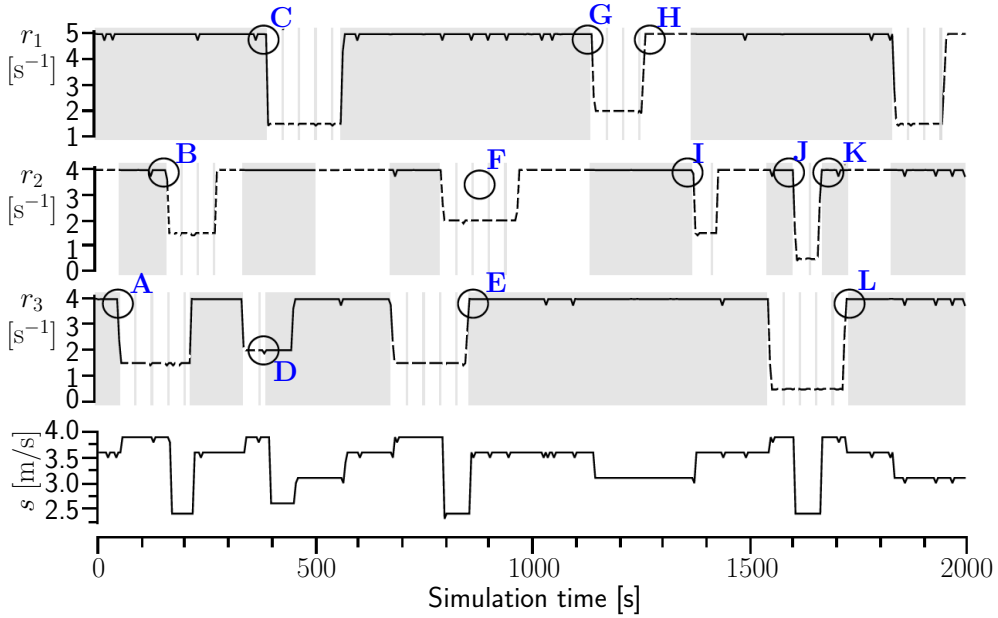


Figure 3.4: Sample pattern of sensor failures and drops in measurement rates for a 3-sensor system, and the sensor configurations and speed chosen by the self-adaptive UUV. Shaded areas correspond to a sensor being switched on, areas not shaded correspond to the sensor being switched off, and thin shaded areas correspond to the system checking whether a sensor has recovered after it experienced problems.

(I) Sensor 1 takes over as sensor 2 is experiencing problems.

(J) With both sensors 2 and 3 in a bad state, the UUV speed must be reduced to a value at which sensor 1 alone can satisfy requirement R1.

(K) The recovery of sensor 2 enables the UUV to continue the mission at higher speed.

(L) When sensor 3 recovers too, it is preferred to sensor 2 as it is more energy efficient.

Given these results, we can safely conclude that RQV successfully managed to drive reconfiguration of a sensor-equipped UUV in the presence of service degradation or complete failure of its sensors. Thus, we consider RQV as a suitable technique for supporting dependable adaptation in UUVs. There are, certainly, other aspects in the UUV domain in which RQV could be useful, but this is subject of future research.

RQ2 (Comparison). To answer this research question, we compared the proposed RQV variants, i.e., those augmented with caching, lookahead, nearly-optimal reconfiguration and combinations thereof, against the standard RQV. Figures 3.5 and 3.6

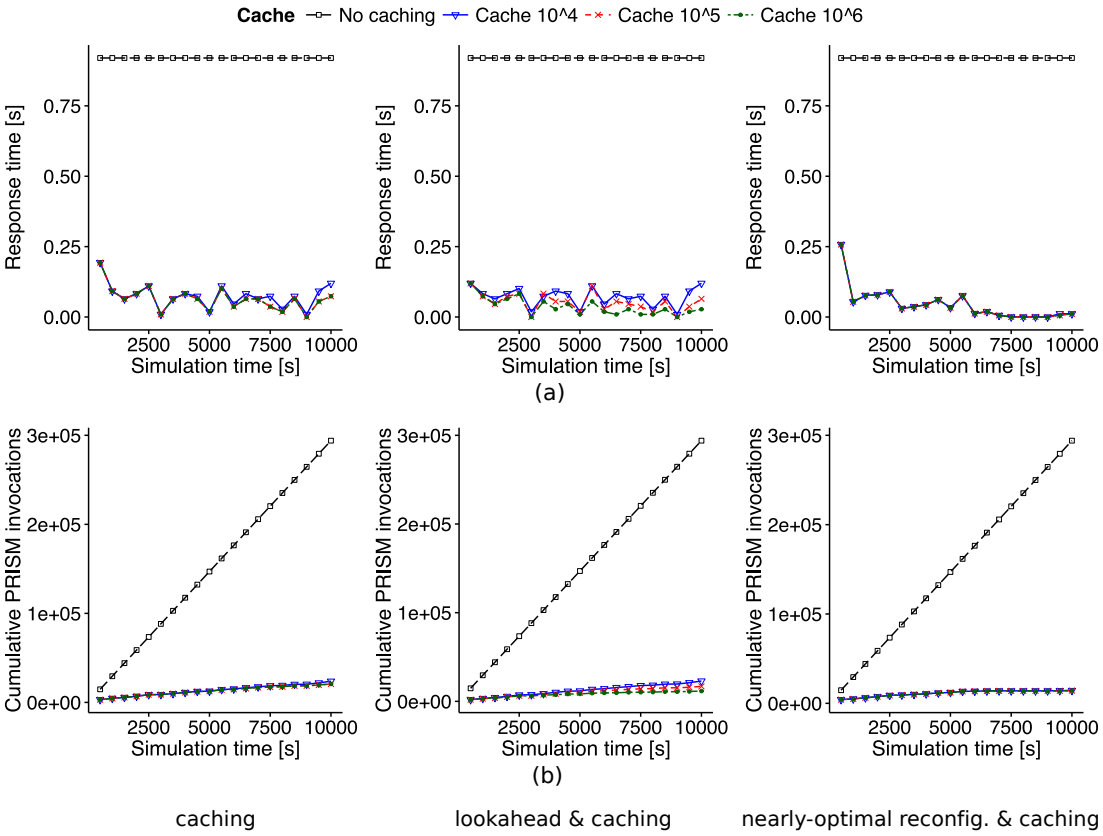


Figure 3.5: Effect of efficient RQV techniques on (a) the average time required to decide a new configuration during an RQV step (*response times* are averaged over 100 RQV steps); and (b) the total number of quantitative verification operations over 2000 RQV steps, for a scenario with low (i.e., $\leq 2\%$) sensor-rate fluctuation during normal operation.

show the RQV response time (i.e., the time to complete all the quantitative verification operations for an RQV step, averaged over successive sequences of 100 steps) and the cumulated number of quantitative verification operations for a 10,000s simulation of the 3-sensor self-adaptive UUV. The pattern of sensor-rate variation described in the previous section (Figure 3.4) corresponds to the first 2,000s of simulated time, and a similar pattern was applied for the remainder of the simulation. In addition to this pattern, the sensor rates for the experiments shown in the two diagrams were also varied during periods when they appear constant in Figure 3.4, by values drawn from a uniform distribution between $[-2\%, 2\%]$ and $[-10\%, 10\%]$ of the maximum rate for these sensors, respectively. Note that the lookahead quantitative verification operations, which are carried outside the actual RQV step, are not included in the results reported in Figures 3.5 and 3.6.

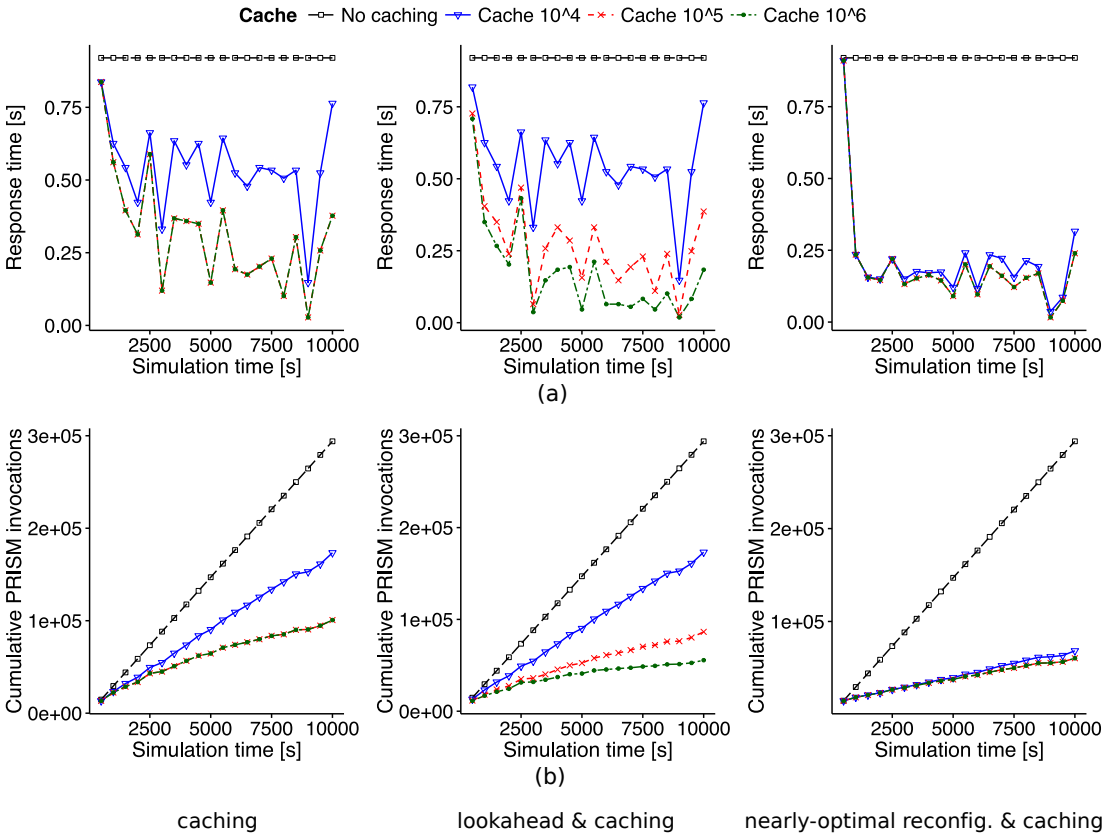


Figure 3.6: Effect of efficient RQV techniques on (a) the average time required to decide a new configuration during an RQV step (*response times* are averaged over 100 RQV steps); and (b) the total number of quantitative verification operations over 2000 RQV steps, for a scenario with high (i.e., $\leq 10\%$) sensor-rate fluctuation during normal operation.

All three techniques and combinations of techniques presented in Figure 3.5 achieved significantly better results than standard RQV. In fact, these techniques carried out an overall number of quantitative verification operations of only 5.2%–7.6% of the number of operations carried out by standard RQV. Clearly, the low sensor-rate variations during periods of normal operation meant that the configurations to be verified were already available in cache or, for the nearly-optimal reconfiguration and caching technique, were similar to configurations seen before. The response time was consistently below 17.5% of the time taken by standard RQV, with the exception of the initial, "learning" period of the nearly-optimal reconfiguration technique. The technique yielded configurations that were on average 28.3% more expensive than the optimal configurations generated by the other approaches. The benefit of a cache larger than 10^4 entries was marginal.

The cache size did make a difference, however, when the sensor-rate variation during

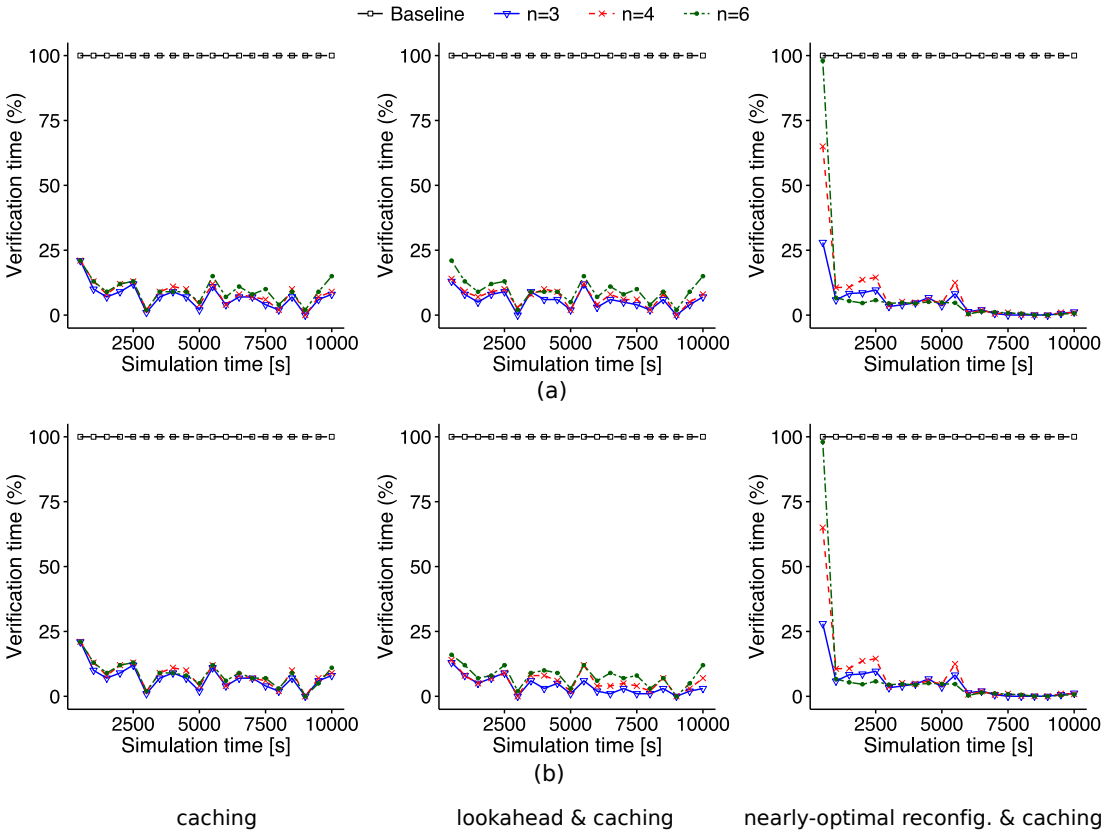


Figure 3.7: Effect of efficient RQV verification on the response time (averaged over 100 RQV steps) for UUV systems with 3, 4 and 6 sensors, low sensor-rate variation during normal operation periods, and using (a) a 10^5-entry cache; and (b) a 10^6-entry cache.

normal operation was higher (Figure 3.6). In this case, the smaller cache size examined (i.e., 10^4 entries) supported the reduction of the overall number of quantitative verification operations to just 58.6% for the caching, and lookahead and caching techniques. The medium cache size, 10^5 entries, achieved reductions to 34.4% and 30.1% for caching, and lookahead and caching, respectively, showing that the cache was insufficient to support effective lookahead. In contrast, the largest cache size, 10^6 entries, supports lookahead, reducing the number of verification operations to only 17.6% when this technique is used in conjunction with caching, compared to also 34.4% when caching alone was used. Nearly-optimal reconfiguration continues to work well, providing reductions to 19–23% of the number of operations for standard RQV, irrespective of cache sizes, but at the expense of using suboptimal configurations. Unsurprisingly, the same pattern is observed in the average response times. The medium sized cache is sufficient for the RQV augmented with caching alone, but not for RQV with lookahead and caching.

One final result that we present is an exploration of the scalability of the introduced techniques. For this purpose, we ran simulations of self-adaptive UUV systems with 3, 4 and 6 sensors, using RQV augmented with the same three techniques and combinations of techniques as above. The resulting average RQV response times, shown in Figure 3.7, are given as percentages of the response times for standard RQV. The results show that very similar benefits are obtained across all system sizes. When caching alone is used, response times of between 1% and 22% of those for standard RQV are obtained with a 10^5-entry cache, and no further improvements are possible when the larger cache is used. Lookahead, however, can take advantage of the larger, 10^6-entry cache, to reduce the response time by a further 6% on average, to between approximately 0.5% and 16% of the standard RQV response time. Nearly-optimal reconfiguration with caching is again doing well irrespective of the cache size, although at the expense of an increased overall cost of 28.3% for $n = 3$, 29.7% for $n = 4$ and 34.2% for $n = 6$.

RQ3 (Insights). We analysed the performance of the evaluated RQV variants with the aim to identify actionable insights regarding the types of systems and adaptation scenarios to which these techniques are applicable. The following criteria were considered: CPU and memory use, response time and solution quality. We present a summary of our findings in Table 3.2.

First, RQV enhanced with caching reduces both CPU usage and the time required to perform an RQV step, but requires additional memory to store recent verification results. The technique is particularly useful in scenarios where a software system experiences low variations in its operating state. In these scenarios, it is possible that verification results corresponding to the current system state already exist in cache. The efficiency of caching, however, is subject to the complexity of the generated keys. As reported in Algorithm 1, a key is partly composed by the values of the parameters associated with a stochastic model. Therefore, in models involving double-valued parameters the complexity depends on the parameter granularity used for generating the keys. As granularity increases, the upper bound of possible keys increases too. As a result, additional memory is required to store the verification results and maintain the same hit ratio.

As far as lookahead is concerned, the technique requires additional CPU power and memory to carry out the pre-computation step and store verification results likely to occur, respectively. In return, lookahead might reduce RQV response time as verification results corresponding to the current system state might have been analysed while system CPU was idle. The technique can cope with higher variations in observable

Table 3.2: Summary of evaluated techniques (compared to standard RQV)

Criterion	Caching	Limited Lookahead	Nearly-Optimal Reconfiguration
CPU use	↓	↑	↓
Memory use	↑	↑	↓
Response time	↓	↓	↓
Solution quality	—	—	↓
Applicability	Low variations in system state	Higher variations than caching, but infrequent major changes	High variations and systems with different model sizes

system parameters than caching, but it resorts to standard RQV when the system is affected by major changes. Since lookahead employs a cache-like mechanism to store the pre-computation results, it also encounters the issues of caching reported above. Furthermore, as a best-effort technique that completes as much as possible of the analysis during idle CPU periods, it is insufficient to use only a large cache. The ϵ parameter should also be set to a meaningful value (for the considered system) so that the technique can extract useful results during the pre-computation step.

Finally, nearly-optimal reconfiguration achieves good performance for systems affected by major changes or experiencing high variation in their observable parameters during normal operation. In fact, the technique reduces both the CPU and memory use, and it is capable of reconfiguring a system much faster than the other proposed RQV variants. These benefits come, of course, at the expense of selecting a suboptimal configuration. Moreover, the utility values *maxUtility* and *minUtility* achieved during the initial learning period should represent reasonably well the utility bounds corresponding to the current system state. Nearly-optimal reconfiguration is suitable in scenarios in which a software system impacted by changes must find quickly a good solution (according to system objectives) and use it to adapt the system. Selecting appropriate values for the parameters associated with the technique (cf. Algorithm 3), i.e., learning period T and *leniency* policy a, requires careful consideration and some domain knowledge. Values close to their upper bounds might cause the technique to operate with increased overheads and reduce its response time, while low values could drive the technique to select expensive configurations.

3.3.4 Threats to Validity

We identified several types of threats that can affect the validity of the study conducted in this chapter. We classify these threats into *construct*, *internal*, and *external* validity.

Construct validity threats may arise because of the procedure followed when designing the UUV case study and any assumptions or simplifications we made. To mitigate this threat, we developed the model of a sensor using information collected from real-world underwater sensors[2]. Concerning the implementation of the UUV system, this is based on MOOS-IvP [20], a widely used platform for developing applications involving UUVs. The platform is not only used for exploring and analysing the behaviour of UUVs in simulated environments, but also for deploying UUVs in real missions (e.g oceanic surveillance and undersea mapping).

Internal validity threats may be due to any bias introduced when obtaining the results and when establishing the cause-effect relationship between the adaptation events and the proposed RQV variants. We limit these threats by evaluating the RQV variants on a test suite comprising several adaptation scenarios of increasing length and complexity. We also performed experiments using UUV systems of different sizes (i.e., number of sensors). Finally, when caching was used, either on its own or in conjunction with another technique, we ran experiments for different cache sizes.

External validity threats can originate from concerns related to the generalisation of our findings. These include difficulties to employ any of the proposed RQV variants as the reconfiguration mechanism in self-adaptive systems. To mitigate this threat, we developed our techniques on top of the standard RQV. Thus, they can be applied to any types of systems supported by standard RQV and can be used with a wide range of probabilistic models and probabilistic temporal logics. Furthermore, our techniques are general enough so that they can be integrated with other state-of-the-art RQV approaches (see Section 2.2.3) to produce more sophisticated RQV variants. Nevertheless, to assess the general applicability of the techniques in other application domains, we need to carry out additional experiments using software systems from these domains.

3.4 Related Work

Caching, limited lookahead and nearly-optimal reconfiguration techniques have been around for a few decades and have been widely applied in engineering software systems. Herein, we are not interested in covering the literature exhaustively. Instead, we aim to

[2]for instance, `http://www.ashtead-technology.com/rental-equipment/rdi-300khz-navigator`

provide a general overview of the applicability of the techniques used in this chapter.

Caching is a common performance optimization technique and is extensively used across all areas of software engineering including networks [213], processors [100], web servers[172], and search engines [90]. Since the early days of computing systems, researchers recognised the advantages of using a small amount of expensive memory to accelerate a slower but larger and less-expensive memory [60, 190]. In processors, for instance, caching has been used to reduce processor-memory traffic [100], while modern multicore computing systems use several levels (L1–L3) of cache memory [43]. Web caching is used to reduce network bandwidth usage, user perceived-delays and server workload [172]. Caching in search engines is primarily employed to improve query throughput and reduce system load [90]. Improving the efficiency of existing *replacement policies* or developing new policies tailored to an application domain is an active area of research [90, 159, 172].

Limited lookahead-style techniques, sometimes called *forward state-space search techniques*, also have a good track record of successful applications in software systems, e.g., for dynamic resource provisioning in computer clusters [144] and speech recognition [170]. Lookahead search in decision-making has been explored to support real-time planning [164] and strategy synthesis in games [165]. The research projects in [184] and [161] conduct theoretical analysis of the technique and study the levels of guarantees that can be obtained for obtaining good solutions in several domains, including the problems of travelling salesman and 0/1-knapsack.

Variants of suboptimal optimisation/configuration have been widely used in optimisation of network traffic [193], malware detection [87], software architecture [209], trajectory generation [131], and many other areas. The research in [193] proposes a near-optimal local search heuristic that performs traffic distribution in networks according to a set of performance criteria. Similarly, [87] presents an approach and supportive tool that automatically analyses behaviours from malware samples and extracts optimally discriminative specifications, which can then be used for malware detection. Finally, [209] introduces an approximation technique, called Filtered Cartesian Flattening, for identifying highly optimal architectural variants that comply with resource constraints and optimise a set of system properties.

This body of research illustrates the applicability of caching, limited lookahead, nearly-optimal reconfiguration and combinations thereof in a wide range of applications. To the best of our knowledge, however, this is the first work that uses these techniques in the context of runtime quantitative verification of self-adaptive software systems.

Subsequently to our publication of the results presented in this chapter, Moreno

et al. [162] proposed a related technique for reconfiguring software systems through combining lookahead and latency awareness. The role of lookahead is similar to our use of the technique, i.e., to project how the system is expected to evolve over a limited horizon. Latency awareness considers the time consumed between making an adaptation decision and actually realising the decision in the target software system. The authors compared their technique against a latency-agnostic feed-forward approach that does not use lookahead. Based on the metrics used, including system utility and response time, the proposed technique outperformed feed-forward. These findings confirm that both limited lookahead and latency awareness improve the effectiveness of adaptation.

3.5 Summary

In this chapter, we adapted three widely used software engineering techniques, namely caching, lookahead and nearly-optimal reconfiguration, to improve the efficiency and overall performance of runtime quantitative verification. When caching is used, verification results from recent reconfigurations are kept into a temporary storage area indexed by a key comprising the observable/configurable parameter values of the parametric model and the index associated with a probabilistic temporal logic formula. If subsequent reconfiguration events involve the verification of a key already in cache, it is sufficient to retrieve these verification results. Limited lookahead uses idle CPU times between successive reconfiguration events to pre-verify system states that could arise in the future. If after a change affecting the system, its current state has already been verified, the verification results are simply retrieved. Finally, nearly-optimal reconfiguration stops the analysis of system configurations as soon as a configuration satisfying the QoS requirements has been found and a "near-optimality" criterion is met.

To evaluate the benefits and limitations of these techniques, we developed a self-adaptive unmanned underwater vehicle (UUV) simulator and carried out experiments involving a wide range of realistic scenarios. The experimental results show that caching can improve the RQV efficiency significantly when there are small variations in the state of the self-adaptive system and its environment during periods of normal operation. When these variations are more significant, lookahead and caching used together achieve much better results, but require a larger cache and depend on the availability of spare computation resources. In contrast, nearly-optimal reconfiguration and caching is less sensitive to cache sizes, and achieves very good performance irrespective of system size, even for high variations in the state of the system during periods of normal operation, at the expense of selecting suboptimal configurations.

Chapter 4

Improving RQV Efficiency Using Evolutionary Algorithms

In the previous chapter, we proposed three conventional software engineering approaches, namely *caching*, *limited lookahead* and *nearly-optimal reconfiguration* that we showed could reduce the overheads of RQV and improve the efficiency of the technique. Notwithstanding the strengths of these approaches, they face difficulties with searching through large configuration spaces to identify configurations that optimise several, and possibly conflicting, QoS requirements (e.g., reliability, performance, cost). This is a non-trivial task whose complexity increases radically with configuration space size and the dependencies/interactions between QoS requirements. Beyond the three proposed techniques, simple and tedious approaches like exhaustive search, trial-and-error, and heuristics are inadequate. Exhaustively searching the configuration space to find an optimal configuration is typically intractable. On the other hand, trial-and-error requires manual verification of numerous alternative instantiations of the system parameters, while heuristics do not generalise well and might be biased towards a particular area of the problem landscape. Even when one of these approaches is in some manner "successful", the identified configurations are often unworthy as they do not represent the optimal trade-offs between the considered QoS requirements.

In this chapter we introduce EvoChecker, a search-based approach that uses evolutionary algorithms (EAs) within the RQV process to address these challenges and to extend the range and size of systems that can be handled by the technique. EA approaches have a long track record of efficiently solving software engineering problems

in highly nonlinear and multidimensional search spaces [9, 42, 74, 86, 155]. EvoChecker exploits the unique abilities of EAs to effectively guide the search towards promising areas of configuration space, to identify Pareto-optimal configurations, and thus to reduce significantly the RQV overheads for reconfiguring a self-adaptive system.

Our EvoChecker search-based approach uses as inputs:

- a *probabilistic model template* that encodes the *configuration parameters* (e.g., alternative architectures, parameter ranges) of the self-adaptive system;
- a set of QoS requirements specifying both constraints (e.g., "At least 95% of the requests must be processed within 200ms") and optimisation objectives (e.g., "The system should minimise energy consumption").

We developed two EvoChecker variants which target different types of self-adaptive systems, in terms of human expert involvement and prioritisation of QoS requirements. First, we developed a *human-in-the-loop* variant whose control loop involves a human operator validating adaptation decisions [41]. This variant employs multi-objective EAs to establish the Pareto front, i.e., the optimal trade-off achievable for the QoS requirements of interest. Given this Pareto front, the operator selects the desired trade-off and the corresponding configuration is used to adapt the system. The second variant is completely automated, but requires a relative prioritisation of the QoS requirements. When the search finishes, the configuration optimising the predetermined trade-off is selected to reconfigure the system. To speed up the search, this variant maintains an archive of configurations used during recent adaptations and "seeds" each EA search for a new configuration with a specific selection of recent configurations from the archive. Despite the potential to use a similar archive in the human-in-the-loop EvoChecker, existing research highlights the main challenges to apply the same principles for multi-objective optimisation [88, 124, 136]. We analyse these challenges later in Section 4.1.4. Therefore, we propose this research direction as future work for the human-in-the-loop EvoChecker (cf. Section 7.2). Finally, it should be noted that the expected adaptation time could be very different between the two EvoChecker variants, typically much larger for the human-in-the-loop variant.

The main contribution of this chapter is the EvoChecker search-based approach, with its *human-in-the-loop* and *automated* variants, which we introduce in Section 4.1. We describe the open-source EvoChecker tool in Section 4.2. Next, in Section 4.3 we present an extensive empirical evaluation of both EvoChecker variants and analyse our findings. We discuss related work and summarise the EvoChecker approach in Sections 4.4 and 4.5, respectively.

4.1 EvoChecker

In line with our work from Chapter 3, EvoChecker uses parametric Markov models to capture both environment uncertainty and system configurations. Thus, *Env* corresponds to the set of possible values for the (observable) parameters of the environment, while *Cfg* is associated with configurable parameters that can be adjusted through the control loop of a self-adaptive system. Let $c = (c_1, c_2, ..., c_k) \in Cfg$ be a system configuration, where $c_i \in V_i, 1 \leq i \leq k$ (i.e., V_i is the value range for the i-th configuration parameter of the system). Let also $e = (e_1, e_2, ...) \in Env$ be an environment state. We assume that any instantiation of $c \in Cfg$ and $e \in Env$ is associated with a valid concrete probabilistic model which can be used to evaluate QoS system attributes of interest.

Foreign Exchange Self-Adaptive System

We will illustrate our approach using a real-world service-based system from the domain of foreign exchange trading that is used by a European foreign exchange brokerage company; for confidentiality reasons we anonymise the system as FX. The FX system, whose Markov model and QoS requirements have been developed as part of this project, implements the workflow in Figure 4.1.

An FX trader can use the system to carry out trades in *expert* or *normal* mode. In the *expert* mode, the trader can provide her objectives or action strategy. FX periodically analyses exchange rates and other market activity, and automatically executes a trade once the trader's objectives are satisfied. In particular, a *Market watch* service retrieves real-time exchange rates of selected currency pairs. A *Technical analysis* service receives this data, identifies patterns of interest and predicts future activity in exchange rates. Based on this prediction and if the trader's objectives are *"satisfied"*, an *Order* service is invoked to carry out a trade; if they are "unsatisfied", execution control returns to the *Market watch* service; and if they are "unsatisfied with high variance", an *Alarm* service is invoked to notify the trader about opportunities not captured by the trading objectives. In the *normal* mode, FX assesses the economic outlook of a country using a *Fundamental analysis* service that collects, analyses and evaluates information such as news reports, economic data and political events, and provides an assessment on the country's currency. If the trader is satisfied with this assessment, she can sell/buy currency by invoking the *Order* service, which in turn triggers a *Notification* service to confirm the successful completion of a trade.

The FX system uses $n_i \geq 1$ functionally equivalent implementations of the i-th service. The j-th implementation, $1 \leq j \leq n_i$ is characterised by its reliability $r_{ij} \in [0, 1]$,

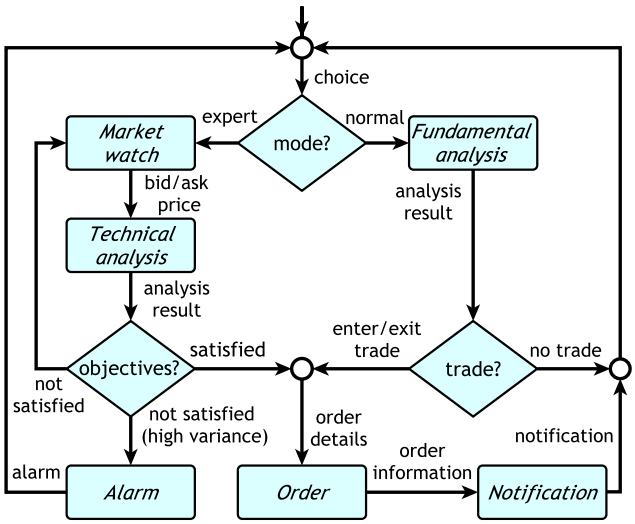


Figure 4.1: Workflow of the FX system.

invocation cost $c_{ij} \in \mathbb{R}^+$ and response time $t_{ij} \in \mathbb{R}^+$.

FX is required to adapt to changes in the observed service implementations reliability r_{ij} and response time t_{ij}, and to complete service failures such that the system satisfies the QoS requirements from Table 4.1. To this end, FX must select one of two invocation strategies by means of a configuration parameter $str_i \in \{\text{PROB, SEQ}\}$: i) a *probabilistic* strategy where one of the implementations is randomly selected based on an FX-specified discrete probability distribution $p_{i1}, p_{i2}, \ldots, p_{in_i}$ (if $str_i = \text{PROB}$); and ii) a *sequential* strategy, where the *enabled* implementations are invoked one after the other based on an execution order, until a successful response is obtained or all invocations fail (if $str_i = \text{SEQ}$). For the SEQ strategy, a parameter $ex_i \in \{1, 2, ..., n_i!\}$ establishes which of the $n_i!$ permutations of the n_i implementations should be used, and a configuration parameter $x_{ij} \in \{0, 1\}$ indicates if implementation j is enabled ($x_{ij} = 1$) or not ($x_{ij} = 0$).

Table 4.1: QoS requirements for the FX system.

ID	Informal description
R1	"Workflow executions must complete successfully with probability at least 0.98"
R2	"The total service response time per workflow execution should be minimised"
R3	"The probability of a service failure during a workflow execution should be minimised"
R4	"The total cost of the third-party services used by a workflow execution should be minimised"

We used the FX system for evaluating the human-in-the-loop and automated EvoChecker variants (Sections 4.3.1 and 4.3.2). To simulate realistic runtime behaviour, we developed simple SOAP-based web services, defined patterns of normal operation and unexpected failures for these services and deployed the services on a single Apache Tomcat web server.

Example 4.1. Suppose that for the FX system, there are three available implementations for the *MarketWatch* service, i.e., $n_1 = 3$. Then, a system configuration c has the general form

$$(str_1, p_{11}, p_{12}, p_{13}, x_{11}, x_{12}, x_{13}, ex_1, \ldots) \in Cfg \tag{4.1}$$

where for the MarketWatch service

- $str_1 \in \{\text{PROB, SEQ}\}$ is the invocation strategy associated with this service;
- $p_{11}, p_{12}, p_{13} \in [0, 1]$ denote the selection probability for the three implementations when the PROB strategy is used;
- $x_{11}, x_{12}, x_{13} \in \{0, 1\}$ signify the enabled service implementations (i.e., those for which $x_{ij} = 1$) when the SEQ strategy is selected;
- $ex_1 \in \{1, 2, ..., 6\}$ indicates which of the $3! = 6$ permutations of the service implementations is used with the SEQ invocation strategy.

An environment state e has the form

$$(r_{11}, t_{11}, r_{12}, t_{12}, r_{13}, t_{13}, ...) \in Env \tag{4.2}$$

where

- $r_{11}, r_{12}, r_{13} \in [0, 1]$ correspond to the (observed) reliability of the three service implementations;
- $t_{11}, t_{12}, t_{13} \in \mathbb{R}^+$ denote their respective response times.

The elements from the system configuration and environment states corresponding to the other five services adopt the same pattern but are shown as ellipses in (4.1) and (4.2) for simplicity.

When dealing with complex self-adaptive systems that have a large configuration space *Cfg*, computing the optimal configuration or the Pareto-optimal configuration set is in most cases infeasible [153]. For instance, this is the case with the FX system, whose configuration space involves real parameters for the probability distributions p_{11}, p_{12}, p_{13} etc. Due to the nonlinearity of stochastic models [145], discretisation of these parameters does not help, as a small change in a state transition probability might cause a large deviation in the values of the QoS attributes. Thus, finding suitable configurations to adapt the system may require prohibitive computational resources or may take an unacceptably long time.

EvoChecker addresses these issues and identifies effective configurations using evolutionary algorithms (EAs). Typical EA examples include genetic algorithms, evolutionary strategies and differential evolution [50]. EAs solve a search problem by assembling a set (i.e., *population*) of solutions (*individuals*) over a number of iterations, such that the population contains better individuals after each iteration. Each individual encodes a candidate solution in the form of a *chromosome*, i.e., a sequence of genes. A *valid* candidate solution is the result of assigning a value to each gene from its value range. In our approach, each gene corresponds to a system configuration parameter $c_i \in V_i, 1 \leq i \leq k$, so a chromosome has the general form $(c_1, c_2, ..., c_k) \in V_1 \times V_2 \times \cdots \times V_k = Cfg$.

Example 4.2. Consider again the FX system with $n_1 = 3$ MarketWatch service implementations. A chromosome denoting a possible system configuration has the structure

$$(str_1, p_{11}, p_{12}, p_{13}, x_{11}, x_{12}, x_{13}, ex_1, \ldots)$$

where $str_1, p_{11}, p_{12}, p_{13}, x_{11}, x_{12}, x_{13}, ex_1$ are the genes associated with the MarketWatch service of the system. The value range for each of these genes is given in Example 4.1. The genes representing the remaining services are omitted in the interest of brevity, but they follow a similar pattern.

4.1.1 Modelling Language

EvoChecker uses the probabilistic model checker PRISM [149] for its verification steps. Accordingly the probabilistic model template is expressed in an extension of the PRISM high-level modelling language. This language is based on the Reactive Modules formalism [8], which describes a system as the parallel composition of a set of *modules*. The state of a *module* is encoded by a set of finite-range local variables, and its state tran-

sitions are defined by probabilistic guarded commands that change these variables, and have the general form:

$$[action]\ guard \text{ -> } e_1 : update_1 + \ldots + e_n : update_n; \tag{4.3}$$

In this command, *guard* is a boolean expression over all the variables in the model. If *guard* evaluates to *true*, the arithmetic expression $e_i, 1 \leq i \leq n$, gives the probability (for discrete-time models, cf. Section 2.1.1.1) or rate (for continuous-time models, cf. Section 2.1.1.2) with which the $update_i$ change of the module variable occurs. The *action* is optional; when present, it forces all modules comprising commands with this *action* to perform one of these commands simultaneously (i.e., to synchronise). For a detailed description of the PRISM modelling language, we refer the reader to the PRISM manual available at `http://www.prismmodelchecker.org/manual`. EvoChecker handles all types of probabilistic models and probabilistic temporal logics supported by PRISM and shown in Table 4.2.

Example 4.3. Figure 4.2 shows an excerpt of the DTMC model of the FX system specified in the PRISM modelling language. The model comprises a WorkflowFX module encoding the workflow of the system and two modules for each service. These two service modules correspond to the probabilistic invocation strategy and the sequential invocation strategy, respectively. Due to limited space, in Figure 4.2 we only show the module associated with the probabilistic strategy of the MarketWatch service. The local variable `state` from the WorkflowFX module (line 3) encodes the state of the system, i.e., which

Table 4.2: Types of models supported by EvoChecker

Type of probabilistic model	**QoS requirement specification logic**
Discrete-time Markov chains	PCTL[a], LTL[b], PCTL*[c]
Continuous-time Markov chains	CSL[d]
Markov decision processes	PCTL[a], LTL[b], PCTL*[c]
Probabilistic automata	PCTL[a], LTL[b], PCTL*[c]
Probabilistic timed automata	PCTL[a]

[a]Probabilistic Computation Tree Logic [22, 111]
[b]Linear Temporal Logic [171]
[c]PCTL* is a superset of PCTL and LTL
[d]Continuous Stochastic Logic [11, 14]

```
1  dtmc

2  module WorkflowFX
3    state: [0..15] init 0; // FX state

4    // Start: expert or normal mode
5    [fxStart] state=0 -> 0.66:(state'=1) + 0.34:(state'=9);

6    //Service #1: Market Watch
7    [startMW] state=1 -> 1.0:(state'=2);
8    [failedMW] state=2 -> 1.0:(state'=5);
9    [succMW] state=2 -> 1.0:(state'=3);

10   //Service #2: Technical Analysis
11   [startTA] state=3 -> 1.0:(state'=4);
12   [failedTA] state=4 -> 1.0:(state'=5);
13   [succTA] state=4 -> 1.0:(state'=6);
14   [resultTA] state=6 -> 0.61 : (state'=1) + 0.28 : (state'=11) + 0.11 : (state'=7);

15   //Service #3: Alarm
16   [startAL] state=7 -> 1.0:(state'=8);
17   [failedAL] state=8 -> 1.0:(state'=5);
18   [succAL] state=8 -> 1.0:(state'=13);

19   //Service #4: Fundamental Analysis
20   [startFA] state=9 -> 1.0:(state'=10);
21   [failedFA] state=10 -> 1.0:(state'=5);
22   [succFA] state=10 -> 0.53 : (state'=0) + 0.27 : (state'=11) + 0.20 : (state'=9);

23   //Service #5: Order
24   [startOR] state=11 -> 1.0:(state'=12);
25   [failedOR] state=12 -> 1.0:(state'=5);
26   [succOR] state=12 -> 1.0:(state'=13);

27   //Service #6: Notification
28   [startNOT] state=13 -> 1.0:(state'=4);
29   [failedNOT] state=14 -> 1.0:(state'=5);
30   [succNOT] state=14 -> 1.0:(state'=15);

31   [failedFX] state=5 -> 1.0:(state'=5);
32   [succFX] state=15 -> 1.0:(state'=15);
33 endmodule

34 const double r11 = 0.998;
35 const double r12 = 0.995;
36 const double r13 = 0.996;

37 const int x11 = 1;
38 const int x12 = 0;
39 const int x13 = 1;

40 const int ex1 = 4;

41 const double p11 = 0.35;
42 const double p12 = 0.45;
43 const double p13= 0.20;

44 // Probabilistic strategy for Service #1: Market Watch
45 module MarketWatchProbStrategy
46   mw: [0..5] init 0; // MW state
47   // Probabilistic service selection
48   [startMW] mw=0 -> p11 : (mw'=1) + p12 : (mw'=2) + p13 : (mw'=3);

49   // Run services
50   [runMW1] mw=1 -> x11 × r11 : (mw'=4) + !x11 || x11 × (1-r11): (mw'=5);
51   [runMW2] mw=2 -> x12 × r12 : (mw'=4) + !x12 || x12 × (1-r12): (mw'=5);
52   [runMW3] mw=3 -> x13 × r13 : (mw'=4) + !x13 || x13 × (1-r13): (mw'=5);

53   // End Market Watch service
54   [succMW] mw=4 -> 1.0:(mw'=0);
55   [failedMW] mw=5 -> 1.0:(mw'=0);
56 endmodule
   ...
```

Reliability of Market Watch service implementations

evolve int x11[0..1];
evolve int x12[0..1];
evolve int x13[0..1];

1

evolve int ex1[1..6];

evolve distribution p1[0.1..0.3][0.3..0.5][0.2..0.6];

2

evolve module MarketWatch

3

Figure 4.2: DTMC model of the FX system; ①–③ represent EvoChecker extensions of the PRISM modelling language.

service is currently invoked, if a service invocation fails, etc. The local variable mw from the MarketWatchProbStrategy module (line 46) records the internal state of the service invocation using the PROB strategy for the MarketWatch service. The WorkflowFX module synchronises with the service invocation module corresponding to the selected invocation strategy through the actions start_, failed_ and succ_, which are associated with scenarios where one or more service implementations are invoked, raise an error or complete execution successfully, respectively. For instance, synchronisation with the MarketWatchProbStrategy module occurs through the actions startMW, failedMW and succMW (lines 7-9). The other FX services operate similarly. Once execution control is given to the module representing a strategy, a service implementation is selected according to the logic of the strategy; e.g., a probabilistic selection is made between the available MarketWatch service implementations (line 48). Then, the selected service is invoked (lines 50-52) and, depending on its observed reliability (lines 34-36), fails to execute (line 55) or completes successfully its task (line 54). In either case, control of execution returns to the WorkflowFX module. If the service executed successfully, FX continues its workflow with the remaining services (lines 9,13,14,18,22,26,30) and concludes its execution (line 32); if a service fails, FX terminates (line 31).

EvoChecker extends the PRISM modelling language with three constructs that support the specification of the possible system configurations from the set *Cfg*, within a *probabilistic model template*. The three constructs are defined below:

1. **Evolvable parameters.** EvoChecker uses the syntax

$$\begin{array}{l}\textsf{evolve int } param\ [min..max];\\ \textsf{evolve double } param\ [min..max];\end{array} \tag{4.4}$$

to declare model parameters of type 'int' and 'double', respectively, and acceptable ranges for them. These parameters can be used in any field of command (4.3) other than *action*, just like constant model parameters declared using '**const int**' and '**const double**' from the original language.

2. **Evolvable probability distributions.** The syntax

$$\textsf{evolve distribution } dist\ [min_1..max_1]\ldots[min_n..max_n]; \tag{4.5}$$

where $[min_i, max_i] \subseteq [0, 1]$ for all $1 \leq i \leq n$, is used to declare an n-element discrete probability distribution, and ranges for the n probabilities of the distribution. The name of the distribution with $1, 2, \ldots, n$ appended as a suffix (i.e., $dist_1$, $dist_2$, etc.) can then be used instead of expressions e_1, e_2, ..., e_n from an n-element command (4.3).

3. **Evolvable modules.** EvoChecker uses the syntax

$$\begin{array}{l} \textsf{evolve module } modName\ implementation_1 \textsf{ endmodule} \\ \ldots \\ \textsf{evolve module } modName\ implementation_n \textsf{ endmodule} \end{array} \tag{4.6}$$

to define $n \geq 2$ alternative implementations of a module *modName*.

The interpretation of the three EvoChecker constructs within a probabilistic model template is described by the following definitions.

Definition 4.1. A valid PRISM probabilistic model augmented with a set of EvoChecker constructs (4.4)–(4.6) is called a *probabilistic model template*.

Definition 4.2. A probabilistic model is an *instance* of a probabilistic model template $\mathcal{T}$ if and only if it can be obtained from $\mathcal{T}$ using the following transformations:

- Each evolvable parameter (4.4) is replaced by a 'const int *param = val;*' or 'const double *param = val;*' declaration (depending on the type of the parameter), where $val \in \{min, \ldots, max\}$ or $val \in [min..max]$, respectively.

- Each evolvable probability distribution (4.5) is removed, and the n occurrences of its name instead of expressions e_1, ..., e_n of a command (4.3) are replaced with values from the ranges $[min_1..max_1]$, ..., $[min_n..max_n]$, respectively. When using a discrete-time model, the sum of the n values is 1.0.

- Each set of evolvable modules with the same name is replaced with a single element from the set, from which the keyword 'evolve' was removed.

Definition 4.3. The set of all probabilistic models that are instances of a probabilistic model template $\mathcal{T}$ represent the configuration space *Cfg*.

Example 4.4. Figure 4.2 illustrates the three EvoChecker constructs employed to transform the DTMC model of the FX system into a probabilistic model template. The replacement of the elements from the shaded dashed rectangles with those from the shaded continuous rectangles show how: ① four evolvable parameters are used to specify the available service implementations for MarketWatch service and their execution order; ② an evolvable distribution is used to specify the transition probabilities associated with the probabilistic selection strategy of the MarketWatchProbStrategy module (line 45); and ③ the module MarketWatch is declared as one of the possible implementations corresponding to the invocation strategies for this service. Note that at least one additional implementation of this module needs to be provided in a valid probabilistic model template. Due to space constraints, the sequential strategy implementation is not included here, but we make it available in Appendix A.

4.1.2 Quality-of-Service Attributes

EvoChecker considers self-adaptive software systems with $n \geq 1$ QoS attributes. An attribute corresponds to a quality facet of a system, typically quantifiable using specific metrics through well-defined processes. For example, the attributes of the FX system that we are interested in comprise the reliability, invocation cost and response time of a workflow execution. Given a system configuration $c \in \mathit{Cfg}$ and an environment state $e \in \mathit{Env}$, an evaluation of the i-th attribute, $1 \leq i \leq n$, can be established by

$$\mathit{attr}_i(e, c) = QV(M(e, c), \Phi_i) \tag{4.7}$$

where:

- M is a probabilistic model parametrised by the environment state and system configuration;
- Φ_i is a probabilistic temporal logic formula (Table 4.2) corresponding to the i-th QoS attribute;
- $QV(..., ...)$ is a function that takes a stochastic model M of a system and a formula Φ_i corresponding to a QoS attribute of the same system, and uses quantitative verification to establish the value of Φ_i for the considered model M. This is

an automatic operation typically performed by *probabilistic* model checkers (e.g., PRISM [149] and MRMC [133]).

Example 4.5. Consider again the FX system and its QoS requirements described informally in Table 4.1. In the parametric DTMC model that we developed for this system (an excerpt of which is depicted in Figure 4.2), the states in which the workflow execution completes successfully are annotated with a "success" label, and a "done" label is associated with the states where the workflow execution terminates (whether successfully or not). We also augment the model with two cost/reward structures. The former, "time" structure associates the time taken by each service implementation to complete its execution with the states that signify the invocation of the service. Likewise, a "cost" structure associates the cost incurred by the invocation of each service implementation with the states modelling these events. Table 4.3 shows how the three QoS attributes used in the system requirements, i.e., workflow reliability, response time and invocation cost, can be formally specified as cost/reward augmented probabilistic temporal logic formulae (Section 2.1.2.1).

Table 4.3: QoS attributes for the FX system

ID	Informal description	Formula Φ_i
$attr_1$	Workflow reliability	$P_{=?}[F\text{“}success\text{”}]$
$attr_2$	Workflow response time	$R^{\text{“time”}}_{=?}[F\text{“}done\text{”}]$
$attr_3$	Workflow invocation cost	$R^{\text{“cost”}}_{=?}[F\text{“}done\text{”}]$

In the following sections, we introduce the two EvoChecker variants developed as part of this thesis. First, we present the human-in-the-loop EvoChecker (Section 4.1.3). Building on this, we then illustrate the fully-automated (and incremental) EvoChecker (Section 4.1.4). For each variant, we demonstrate i) how it formalises the system QoS requirements using the n QoS attributes; ii) how it formalises the problem of extracting Pareto-optimal configurations; and iii) how it employs EAs within the RQV process to identify a set of these Pareto-optimal configurations.

4.1.3 Human-in-the-Loop EvoChecker

Incorporating human input into the control loop of self-adaptive systems has been advocated by recent research [41]. This could be advantageous when it is difficult to determine a priori the best trade-off between multiple QoS requirements, when the system uses expensive equipment or when the mission undertaken by the system is highly critical [50]. Once the human-in-the-loop EvoChecker completes the search, it produces a set of equally good configurations for adapting a system. A human then acts as a sophisticated system-level decision maker (with better insight regarding the best configuration) who analyses the configurations and selects the most desirable configuration for adapting the system.

Formalising System QoS Requirements

The human-in-the-loop EvoChecker variant supports the specification of QoS requirements for a software system by employing the $n \geq 1$ QoS attributes. These requirements are classified into $n_1 \geq 0$ *constraints* and $n_2 \geq 1$ *optimisation objectives*. *Constraints* define bounds for the acceptable values of some of the n QoS attributes, while *optimisation objectives* specify QoS attributes that should be minimised or maximised (subject to all constraints being satisfied). Without loss of generality, we will assume that the QoS attributes should be minimised. Formally, a software system considered by this EvoChecker variant needs to satisfy $n_1 \geq 0$ *constraints* of the form

$$R_i^C : \ attr_i(e,c) \bowtie_i bound_i, \ 0 \leq i \leq n_1, \tag{4.8}$$

where

- $\bowtie_i \in \{<, \leq, \geq, >, =\}$;
- $bound_i \in \mathbb{R}$ is an acceptable bound that must be met by the i-th QoS attribute

and optimise $n_2 \geq 1$ *objectives* of the form

$$R_i^O : \ \mathsf{minimise} \ attr_i(e,c), \ n_1 + 1 \leq i \leq n_1 + n_2 \tag{4.9}$$

Example 4.6. The QoS requirements of the FX system (Table 4.1) comprise one constraint (R1) and three optimisation objectives (R2–R4). Table 4.4 shows the formalisation of these requirements in terms of the QoS attributes from Table 4.3.

Table 4.4: Formal specification of QoS requirements for the FX system

ID	Formal description	Type
R1	$attr_1(e,c) \geq 0.98$	constraint (4.8)
R2	minimise $attr_2(e,c)$	objective (4.9)
R3	minimise $1 - attr_1(e,c)$	objective (4.9)
R4	minimise $attr_3(e,c)$	objective (4.9)

Formalising The Reconfiguration Problem

The human-in-the-loop EvoChecker aims to identify as many as possible of the configurations that capture the optimal trade-offs between the n_2 objectives, subject to satisfying the n_1 constraints. To this end, we employ Pareto optimality theory [67] to establish the Pareto front for the n_2 objectives, i.e., the optimal trade-off surface of these objectives. Once the Pareto front is determined, a system expert (i.e., the human in the loop) can select the available configuration with the most suitable trade-off and the corresponding configuration can be used to reconfigure the system. According to the classification of techniques for solving multiobjective optimisation problems by Coello et al. [50], this is an *a posteriori* variant since the searching process occurs before any preference specification is carried out.

Consider a self-adaptive software system with configuration space *Cfg*, given by a probabilistic model template $\mathcal{T}$, and a set of QoS requirements that consists of $n_1 \geq 0$ constraints and $n_2 \geq 1$ optimisation objectives. Given the current environment state $e \in Env$, the *reconfiguration problem* involves finding the *Pareto-optimal set PS* of configurations from *Cfg* that satisfy the n_1 constraints and are *non-dominated* with respect to the n_2 optimisation objectives:

$$PS = \{c \in Cfg \mid (\forall 0 \leq i \leq n_1 \bullet attr_i(e,c) \bowtie_i bound_i) \wedge (\nexists c' \in Cfg \bullet c \prec c')\} \tag{4.10}$$

with the *dominance relation* $\prec : Cfg \times Cfg \rightarrow \mathbb{B}$ (assuming minimisation of QoS objectives) defined by

$$\begin{aligned} \forall c, c' \in Cfg \bullet c \prec c' \equiv \quad & \forall\, n_1 + 1 \leq i \leq n_1 + n_2 \bullet attr_i(e,c) \leq attr_i(e,c') \wedge \\ & \exists\, n_1 + 1 \leq i \leq n_1 + n_2 \bullet attr_i(e,c) < attr_i(e,c'). \end{aligned}$$

Finally, given the Pareto-optimal set PS, the *Pareto front* PF is defined by

$$PF = \{(a_{n_1+1}, a_{n_1+2}, \ldots, a_{n_1+n_2}) \in \mathbb{R}^{n_2} \mid \exists c \in PS \bullet \forall n_1 + 1 \leq i \leq n_1 + n_2 \bullet a_i = attr_i(e, c)\}, \tag{4.11}$$

so that the system expert can make informed decisions considering the trade-offs between QoS objectives (alongside any domain knowledge) when choosing between the configurations within the set PS.

Approach Description

The human-in-the-loop EvoChecker finds a set of effective configurations that closely approximates this Pareto-optimal set using evolutionary algorithms for multiobjective optimisation (e.g., NSGA-II [57], SPEA2 [221], MOCell [166]). The high-level architecture of our human-in-the-loop EvoChecker approach is depicted in Figure 4.3. We assume that a system engineer produces a probabilistic model template T that describes the behaviour of the system and contains instances of the three 'evolvable' constructs (Section 4.1.1). The template is given to a *Template parser* component that generates a parametric probabilistic model M. This model is supplied to a *Monitor* component used by our approach. The parser also uses the encoding rules from Table 4.5 to extract the configuration parameters $c_1, c_2, ..., c_k$ and to construct the configuration space *Cfg*. Using the appropriate probabilistic temporal logic (e.g., PCTL described in Section 2.1.2.1 for the FX system), the n QoS system attributes are formalised into mathematical formulae $\Phi_1, ..., \Phi_n$. These formulae are used by an *Individual analyser* component of our solution. Finally, the n_1 constraints and the n_2 objectives are specified according to (4.8) and (4.9), respectively. These elements along with the configuration space *Cfg* are used to initialise the *Multi-objective evolutionary algorithm* component that plays the main role in our approach.

At runtime, the *Monitor* obtains the current environment state $e \in Env$ through *Sensors*, either periodically or after relevant changes. This component produces an updated model M' in which the parameters associated with the state of the environment are fixed. The *Multi-objective evolutionary algorithm*, then, creates a random initial population and starts the search for Pareto-optimal solutions using a standard EA approach, as summarised next.

This search involves evaluating different individuals (i.e., potential new system configurations) through invoking an *Individual analyser* component that takes as inputs an

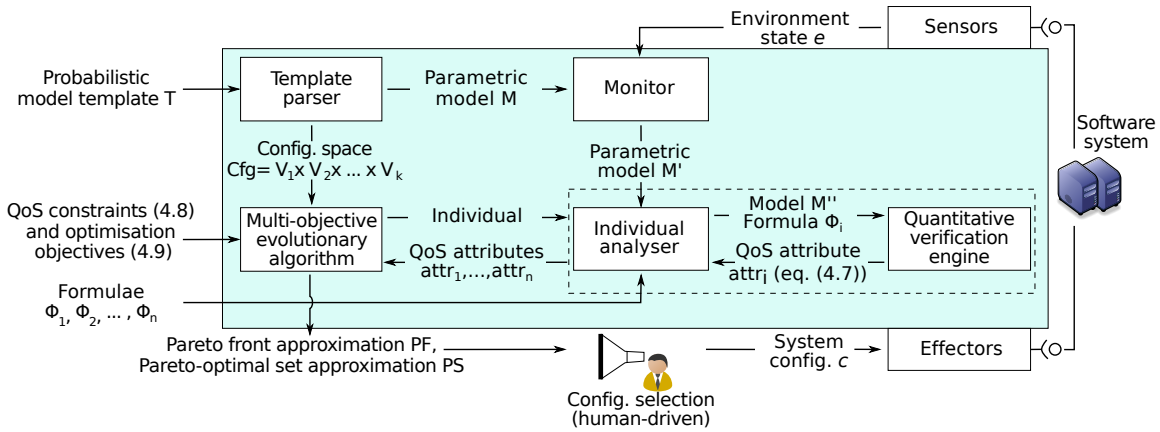


Figure 4.3: High-level human-in-the-loop EvoChecker architecture.

individual and the updated parametric model M'. This component combines the two inputs into a valid probabilistic model M'' in which the remaining parameters from M' are fixed using the values from the genes of the analysed individual. The *Individual analyser* then invokes a *Quantitative verification engine* to determine the attributes $attr_i$, $1 \leq i \leq n$, corresponding to the analysed system configuration. These attributes are used by the *Multi-objective evolutionary algorithm* to establish whether the analysed individual satisfies the n_1 QoS constraints. For each individual that satisfies these constraints, the component determines the values associated with the n_2 QoS objectives.

Once all individuals have been evaluated, the *Multi-objective evolutionary algorithm* performs an *assignment*, *reproduction* and *selection* step. During *assignment*, the algorithm establishes the fitness of each individual (e.g., its rank in the population, the number of individuals that dominate it or the number of individuals which it dominates). Fit individuals have higher probability to enter a "mating" pool and to be chosen for reproduction and selection. With *reproduction*, the algorithm creates new individuals from the mating pool by means of *crossover* and *mutation*. *Crossover* randomly selects two fit individuals and exchanges genes between them to produce offspring with potentially higher fitness values. *Mutation*, on the other hand, introduces variation in the population by selecting an individual from the pool and creating an offspring by randomly changing a subset of its genes. Finally, through *selection*, a subset of the individuals from the current population and offspring becomes the new population that will evolve in the next generation.

The *Multi-objective evolutionary algorithm* uses *elitism*, a strategy that directly propagates into the next population a subset of the fittest individuals from the current population. This strategy ensures the iterative improvement of the Pareto-optimal approximation set PS and the Pareto front approximation PF. Elitism also guarantees

Table 4.5: EvoChecker gene encoding rules

Evolvable feature of the probabilistic model template	**Type**	**EvoChecker gene(s) Cardinality**	**Value range V_i**
evolve int $param[min..max]$;	int	1	$\{min,...,max\}$
evolve double $param[min..max]$;	double	1	$[min..max]$
evolve distribution $dist[min_1..max_1]\ldots$ $\ldots[min_n..max_n]$;	double	n	$[min_1..max_1]$ $\ldots[min_n..max_n]$
evolve module *mod implementation*$_1$ endmodule $\ldots$ evolve module *mod implementation*$_m$ endmodule	int	1	$\{1, 2, ..., m\}$

convergence, meaning that if the global optimum (e.g., the Pareto front) is discovered, the population will eventually converge to that optimum. Furthermore, the multi-objective EAs used by EvoChecker maintain diversity in the population and generate a Pareto-optimal approximation set spread as uniformly as possible across the search space. To achieve this, they use algorithm-specific mechanisms suitable for diversity preservation, e.g., by combining the nondomination level of each evaluated individual and the population density in its area of the search space, by grouping individuals into neighbourhoods that share their fitnesses, etc[1].

The evolution of fitter individuals continues until one of the following termination criteria is met: i) the allocated computation time is exhausted; ii) the maximum number of individual evaluations has been reached; or iii) no improvement in the quality of the best individuals has been detected over a predetermined number of successive iterations. When the evolution terminates, the set of nondominated individuals, i.e., the first front, is used to construct the Pareto-optimal approximation set PS and the set of QoS attributes associated with each individual is used to assemble the Pareto front approximation PF. If the number of optimisation objectives $n_2 \leq 3$, the Pareto front approximation PF can be depicted graphically [194]; e.g., Figures 4.8 and 4.9. This graphical representation enables visual inspection and decision making. In systems where $n_2 > 3$, visual inspection is possible only after fixing some of these objectives to specific values. Front analysis can be also performed by presenting the fronts in a matrix-based format. A system expert can then analyse the PS and PF sets and select a configuration $c \in PS$ to reconfigure the software system.

[1] For example, NSGA-II [57] associates a nondominance level of 1 to all nondominated individuals of a population, a level of 2 to the individuals that are not dominated when level-1 individuals are ignored etc. Individuals not satisfying problem constraints receive a default level of ∞. SPEA2 [221] evaluates population density as the inverse of the distance to the k-th nearest neighbour of the individual.

4.1.4 Automated EvoChecker

Similarly to the human-in-the-loop EvoChecker, the automated EvoChecker[2] runs within the closed control loop of an RQV-driven self-adaptive system. However, the incremental EvoChecker requires a relative prioritisation of the QoS objectives before starting the search and does not involve human input to reconfigure the system after it starts executing. This prioritisation can be in the form of a weight (i.e., a nonnegative constant) assigned to each objective, by specifying an ordering for evaluating the objectives etc. When the system fails to satisfy its QoS requirements or operates poorly, automated EvoChecker uses this prioritisation to find a new configuration that restores the compliance and improves the state of the system.

To speed up the search for a new configuration, the automated EvoChecker builds on the principles of incrementality (cf. Section 2.2.3.1) and exploits the fact that changes in a self-adaptive system are typically localised [95]. As reported in other domains [123, 135], and also discussed in related work (Section 4.4), an effective initialisation of the EA search can improve its convergence and can yield better-quality solutions. The key idea is to maintain an *archive* of effective configurations identified during recent reconfigurations of the self-adaptive system and to "seed" each EA search for a new configuration with a subset of these recent configurations, which encode solutions to potentially similar adverse events that the self-adaptive system experienced in the past.

Integrating the incrementality principles within the human-in-the loop EvoChecker to generate the Pareto-optimal approximation set and the Pareto front approximation is a possible extension for this variant. Nevertheless, research in this direction is limited [135] and the effect of its use in large-scale optimisation problems is open to dispute [136]. Furthermore, maintaining an archive with several Pareto-optimal approximation sets from recent reconfigurations requires not only additional memory but also specific techniques for clustering the individuals and selecting the most suitable for seeding a new search. Most of the existing work (also discussed later in Section 4.4), uses as a seed a subset of the Pareto-optimal set such as the corner cases (i.e., individuals that ignore all but one optimisation objective) [53, 118] or linear combinations (i.e., individuals whose objective values correspond to specific weights) [88, 124]. Despite the common agreement about the usefulness of seeding, recent research fails to provide solid justification why this holds for a combination of EA and fitness landscape or to suggest any practical guidelines [88]. This is an interesting research question and needs further investigation. Therefore, we propose this research as future work (see Section 7.2).

[2] As we explain here, we also call this EvoChecker variant "incremental"; the two names are used interchangeably throughout the thesis.

Formalising System QoS Requirements

The incremental EvoChecker, like its human-in-the-loop counterpart, uses the $n \geq 1$ QoS attributes to formalise the QoS requirements of a software system. From these requirements, $n_1 \geq 0$ are *constraints*. System compliance with the i-th constraint, $0 \leq i \leq n_1$, depends on the value of its i-th QoS attribute, given by (4.8).

The remaining requirements are expressed as a cost. A cost function specifies how some or all of the n QoS attributes should be optimised. This function captures the relation between the n QoS attributes and helps differentiating between multiple system configurations satisfying the n_1 QoS constraints. We define cost as a function $cost : X_1 \times X_2 \times \ldots \times X_n \to \mathbb{R}_+$, where $X_i, 1 \leq i \leq n$, is the value range associated with the i-th attribute. Thus, for any $(e, c) \in Env \times Cfg$

$$cost(attr_1(e,c), attr_2(e,c), ..., attr_n(e,c)) \in \mathbb{R}_+ \tag{4.12}$$

represents the cost corresponding to the evaluation of the n QoS attributes.

Possible examples of *cost* functions include lexicographic ordering, criterion-based, ϵ-constrained and aggregation-based (e.g., linear and nonlinear) functions [50].

Example 4.7. Consider again the QoS requirements of the FX system from Table 4.1. Requirement R1 is the only constraint (4.8), while requirements R2–R4 are used to produce the cost function (4.12). Assuming an environment state $e \in Env$ and a linear aggregating function in which weights $w_1, w_2, w_3 > 0$ express the desired trade-off between the three QoS attributes, the constraint and cost functions are formally specified in Table 4.6.

Table 4.6: Formal specification of QoS requirements for the FX system

ID	Formal description	Type
R1	$attr_1(e,c) \geq 0.98$	constraint (4.8)
R2–R4	minimise $w_1/attr_1(e,c) + w_2 attr_2(e,c) + w_3 attr_3(e,c)$	cost (4.12)

Formalising the Reconfiguration Problem

In incremental EvoChecker, the desired trade-off between the objectives is specified beforehand using the cost function (4.12). Therefore, the target is to identify the optimal configuration that satisfies the n_1 constraints and minimises the system cost. Depending on the specification of the cost function and compared to all other solutions identified during the search, the selected solution is Pareto optimal [50]. Referring again to the classification of techniques for multiobjective optimisation problems [50], the automated EvoChecker belongs to the *a priori* group since the importance of objectives is specified prior to search.

Let *Cfg* be the configuration space of a self-adaptive software system, given by the probabilistic model template $\mathcal{T}$. The system QoS requirements comprise $n_1 \geq 0$ constraints (4.8) and a cost (4.12). Given the current environment state $e \in Env$, the *reconfiguration problem* involves finding a configuration $c \in Cfg$ such that

$$\begin{aligned} &\forall\, 0 \leq i \leq n_1 \bullet attr_i(e,c) \bowtie_i bound_i \wedge \\ &\quad \forall c' \in Cfg \bullet (\forall\, 0 \leq i \leq n_1 \bullet attr_i(e,c') \bowtie_i bound_i) \implies \\ &\qquad cost(attr_1(e,c), attr_2(e,c), ..., attr_n(e,c)) \leq \\ &\qquad\quad cost(attr_1(e,c'), attr_2(e,c'), ..., attr_n(e,c')) \end{aligned} \tag{4.13}$$

Approach Description

Adverse events change the state of the environment and invalidate an existing configuration, triggering the search for a new effective configuration at runtime. Some of these environment states have similar impact on system components while others are likely to occur multiple times during system operation [95]. Our incremental EvoChecker aims to exploit this characteristic of real-world systems, and in particular, to exploit the knowledge that can be gained from recent reconfigurations of a self-adaptive system. To this end, we maintain an *archive* of configurations generated during recent adaptations and use this archive to seed the initial population of a new EA search.

Starting with an empty archive, the incremental EvoChecker iteratively builds the archive used for the next reconfiguration using an *archive updating strategy*. This strategy selects configurations from the final EA population associated with the current event. The configurations within this population are ordered according to a preference relation that comprises the following criteria: ① an individual that meets all n_1 constraints is preferred over an individual that violates one or more constraints; ②

for any two individuals that satisfy all constraints, the individual with the lowest cost is preferred; and ③ for any two individuals that both violate at least one constraint, the individual with the lowest overall violation is preferred. The ordered configuration set (with the preference relation), the *archive* and the *archive updating strategy* σ are formally described by the following definitions.

Definition 4.4. *Let $e \in Env$ be an environment state that affected a self-adaptive system. Let also $violation : Env \times Cfg \to \mathbb{R}_+$ be a function that quantifies the level of violation of the n_1 QoS constraints for each combination $(e, c) \in Env \times Cfg$. Then, a set of configurations created by an EA in response to state e is a totally ordered set $C_e \subseteq Cfg$ where*

$$
\begin{aligned}
&\forall c, c' \in C_e \bullet c < c' \equiv \\
&\quad \forall 1 \leq i \leq n_1 \bullet attr_i(e,c) \bowtie_i bounds_i \wedge \exists 1 \leq i \leq n_1 \bullet \neg(attr_i(e,c') \bowtie_i bounds_i) \bigvee && ① \\
&\quad \forall 1 \leq i \leq n_1 \bullet attr_i(e,c) \bowtie_i bounds_i \ \wedge \ attr_i(e,c') \bowtie_i bounds_i \ \wedge && ② \\
&\qquad\qquad cost(attr_1(e,c), attr_2(e,c), ..., attr_n(e,c)) < \\
&\qquad\qquad\quad cost(attr_1(e,c'), attr_2(e,c'), ..., attr_n(e,c')) && \bigvee \\
&\quad \exists 1 \leq i,j \leq n_1 \bullet \neg(attr_i(e,c) \bowtie_i bounds_i) \ \wedge \ \neg(attr_j(e,c') \bowtie_j bounds_j) \ \wedge && ③ \\
&\qquad\qquad violation(e,c) < violation(e,c')
\end{aligned}
$$

Definition 4.5. *Let $C_e \subseteq Cfg$ be the (ordered) set of configurations created in response to a change in the environment state $e \in Env$ and $Arch$ be the archive before the change. Then an archive updating strategy is a function $\sigma : Cfg \to \mathbb{B}$ such that the archive of configurations for the next environment state $e' \in Env$ is calculated as the set*

$$Arch' = \{c \in Arch \cup C_e \mid \sigma(c)\} \tag{4.14}$$

Depending on the criteria specified for revising the archive, there are several possible implementations for the archive updating strategy σ. For instance, a strategy could be: i) *prohibitive*, i.e., maintain an empty archive; ii) *selective*, i.e., use a subset of the best configurations from the current adaptation step; or iii) *complete*, i.e., use the entire set of configurations from the current adaptation step. The general principle, though, is that a configuration is "worthy" of inclusion in the archive if it could provide extra knowledge and help to solve faster the reconfiguration problem (4.13).

We formally define below four different archive updating strategies that we used to evaluate the incremental EvoChecker (Section 4.3). Let $position : C_e \rightarrow \{1, 2, \ldots, |C_e|\}$ be a function that indicates the position of a configuration $c \in C_e$, i.e., $position(c) = |c' \in C_e \setminus \{c\} | c' < c| + 1$.

A *prohibitive strategy* does not retain any configurations in the archive, given by

$$\sigma(c) = false, \forall c \in Arch \cup C_e \tag{4.15}$$

A *complete recent strategy* uses the entire population of configurations from the current adaptation step and removes the previous configurations from the archive, given by

$$\sigma(c) = \begin{cases} true, & \text{if } c \in C_e \\ false, & \text{otherwise} \end{cases} \tag{4.16}$$

A *limited recent strategy* keeps the $x, 0 \leq x \leq |C_e|$, best configurations from the current adaptation step and removes the previous configurations from the archive, given by

$$\sigma(c) = \begin{cases} true, & \text{if } c \in C_e \text{ and } position(c) \leq x \\ false, & \text{otherwise} \end{cases} \tag{4.17}$$

A *limited deep strategy* accumulates the $x, 0 \leq x \leq |C_e|$ best configurations from all previous adaptation steps, given by

$$\sigma(c) = \begin{cases} true, & \text{if } c \in C_e \text{ and } position(c) \leq x \\ true, & \text{if } c \in Arch \\ false, & \text{otherwise} \end{cases} \tag{4.18}$$

Note that the *limited deep strategy* yields archives that grow in size after each adaptation step. If the archive size exceeds the initial size of the EA population, a subset of the individuals from the archive must be chosen to seed the new EA search. Possible methods to deal with this include i) discarding the least recently used individuals (in which case the individuals must be timestamped); or ii) performing a random selection.

The high-level architecture of the incremental EvoChecker is shown in Figure 4.4. We briefly present the overall approach but focus on the elements distinct from the human-in-the-loop EvoChecker. A *Template parser* uses a probabilistic model template T and the encoding rules from Table 4.5 to extract the configuration space *Cfg* and the parametric model M. The n QoS attributes are formalised into formulae $\Phi_1, \Phi_2, \ldots, \Phi_n$

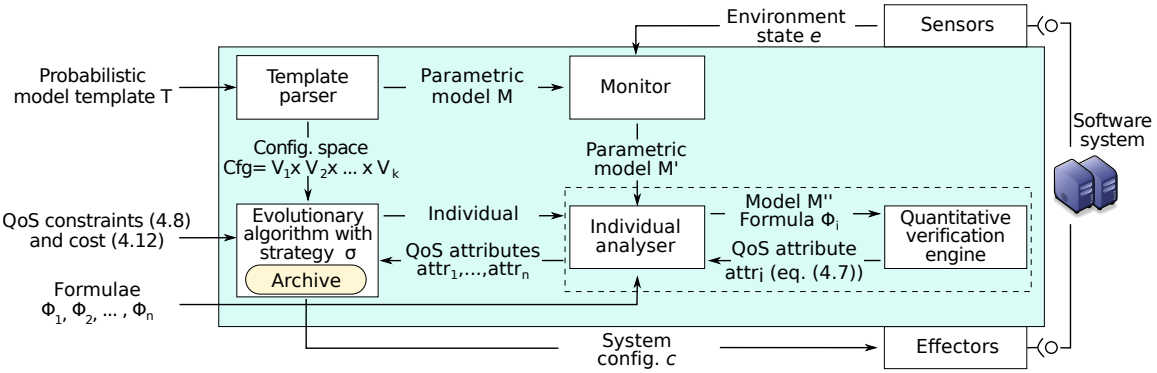


Figure 4.4: High-level automated EvoChecker architecture.

using the probabilistic temporal logic for model M, while the specifications for the n_1 QoS constraints and the system cost conform to (4.8) and (4.12), respectively.

While the system is operating, the current environment state $e \in Env$ is provided to a *Monitor* component, which in turn updates the parametric model M' by assigning values to the parameters corresponding to the state of the environment. Before starting the search for a new effective configuration, the *Evolutionary algorithm* uses the *Archive* *Arch* to create the initial EA population. To this end, configurations within the archive are imported into the population. If the population is not complete, new individuals are generated randomly.

The *assignment*, *reproduction* and *selection* steps are similar to the human-in-the-loop EvoChecker. For each individual, an *Individual analyser* component extracts its gene values and produces a valid probabilistic model M'' which is then used to establish the values of the n QoS attributes. The *Evolutionary algorithm* uses this information to identify whether an individual complies with the n_1 constraints, and if this holds, the associated cost is calculated. Next, an offspring is created in two steps: i) an *assignment* step assigns a fitness value to each individual, which increases the likelihood for choosing individuals with good fitness; and ii) a *reproduction* step creates new individuals using *crossover* and *mutation*. The elitist *Evolutionary algorithm* then performs a *selection* step through which it generates the new population that will evolve in the next generation, using a subset of the individuals from the current population and offspring.

When one of the termination criteria is satisfied, the EA stops executing. The best individual from the final population is the solution to the optimal reconfiguration problem (4.12) and is used to adapt the software system. Finally, incremental EvoChecker employs an archive updating strategy σ, e.g., (4.15)–(4.18), to select some configurations from the last population and build the archive $Arch'$ (4.14) for the next EA search.

4.2 Implementation

We automated the EvoChecker approach by implementing a tool that supports both the human-in-the-loop and the automated EvoChecker variants. To this end, the *Quantitative verification engine* and the *(Multi-objective) Evolutionary algorithm* components integrate the probabilistic model checker PRISM [149] and the Java-based framework for multi-objective optimization with metaheuristics jMetal [66], respectively. We developed the remaining components in Java, using the Antlr[3] parser generator to build the *Template parser*, and implementing the *Individual analyser*, *Monitor*, *Sensor*, and *Effector* components specifically for the EvoChecker tool. The open-source code of EvoChecker, the full experimental results summarised in the following section, additional information about EvoChecker and the case studies used for its evaluation are available at `http://www-users.cs.york.ac.uk/~simos/EvoChecker`.

4.3 Evaluation

We performed extensive experiments to evaluate the effectiveness of the two EvoChecker variants. Among the techniques comprising the EAs family (e.g., genetic algorithms, evolution strategies, differential evolution), we used genetic algorithms (GAs) to realise the *Evolutionary algorithm* component from Figures 4.3 and 4.4. We made this decision due to the proven competence of GAs in the area of search-based software engineering [9, 175]. See also Section 4.4 for a discussion of related work. As we explain later in more details, for the human-in-the-loop EvoChecker we use multi-objective genetic algorithms (MOGAs), while for the automated EvoChecker we employ a single objective (generational) GA. Experimenting with other EA types is outside the scope of this thesis.

In Sections 4.3.1 and 4.3.2, we describe the evaluation procedure and the results obtained for the human-in-the-loop and automated EvoChecker, respectively. For each variant, we introduce the research questions that guided the experimental process, we describe the experimental setup including the self-adaptive software systems used in the evaluation, we illustrate the methodology followed for obtaining and analysing the results, and finally we present and discuss our findings. We conclude the evaluation with a review of threats to validity (Section 4.3.3).

[3] `http://www.antlr.org`

4.3.1 Human-in-the-Loop EvoChecker Evaluation

4.3.1.1 Research Questions

The aim of our experimental evaluation was to answer the following research questions.

RQ1 (Validation): How does human-in-the-loop EvoChecker perform compared to random search? We used this research question to establish if this EvoChecker variant "comfortably outperforms a random search" [116], as expected of effective search-based software engineering solutions.

RQ2 (Comparison): How do human-in-the-loop EvoChecker instances using different MOGAs perform compared to each other? Since we devised human-in-the-loop EvoChecker to work with any MOGA, we examined the results produced by EvoChecker instances using three established such algorithms (i.e., NSGA-II [57], SPEA2 [221], MOCell [166]).

RQ3 (Insights): Can human-in-the-loop EvoChecker provide insights into the trade-offs between the QoS attributes of alternative software architectures and configurations? To support system experts in their decision making, EvoChecker must provide insights into the trade-offs between multiple QoS objectives. To address this question, we identified a range of decisions suggested by the EvoChecker results for the software systems considered in our evaluation.

4.3.1.2 Experimental Setup

The experimental evaluation comprised multiple scenarios associated with two software systems from different application domains. We evaluated the human-in-the-loop EvoChecker on a software-controlled dynamic power management (DPM) system adapted from [174, 188] and described in Appendix B, and on the foreign exchange (FX) self-adaptive service-based system described in Section 4.1.

We performed a wide range of experiments using the system instances from Table 4.7. The column 'Details' reports the capacity of the two request queues ($Qmax_H$ and $Qmax_L$) and the number of power managers available ($m = 2$) for the DPM system; and the number of third-party implementations for each service of the FX system[4]. The

[4]The $n = 8$ services used by FX_Large correspond to using two-part composite service implementations for the Technical analysis and Fundamental analysis services from Figure 4.1

Table 4.7: Analysed system variants for the human-in-the-loop EvoChecker

Variant	Details	Size	$\mathbf{T_{run}}[s]$
DPM_Small	$Qmax_{H,L} \in \{1, ..., 10\}$, $m = 2$	2E+14	0.1050
DPM_Medium	$Qmax_{H,L} \in \{1, ..., 15\}$, $m = 2$	4.5E+14	0.2118
DPM_Large	$Qmax_{H,L} \in \{1, ..., 20\}$, $m = 2$	8E+14	0.3796
FX_Small	$n_1 = \cdots = n_4 = 3, n_5 = n_6 = 1$	8.49E+31	0.0858
FX_Medium	$n_1 = \cdots = n_6 = 4$	2.05E+65	0.1695
FX_Large	$n_1 = \cdots = n_8 = 4$	1.21E+87	0.4162

column 'Size' lists the configuration space size assuming a two-decimal points discretisation of the real parameters and probability distributions of the probabilistic model template (cf. Table 4.5). Given the nonlinearity of most probabilistic models, this is the minimum precision we could assume as an 0.01 increase or decrease in one of these parameters can have a significant effect in the evaluation of a QoS attribute. Finally, the column 'T_{run}' shows the average running time per system variant for evaluating a configuration. Note that the EvoChecker run time depends on the size of model $\mathcal{M}$ and the time consumed by the probabilistic model checker to establish the $n1 + n2$ QoS attributes from (4.7) and on the computer used for the evaluation.

We conducted a two-part evaluation for the human-in-the-loop EvoChecker. First, to assess the stochasticity of this EvoChecker variant when different MOGAs are adopted and also to eliminate the possibility that any observations may have been obtained by chance, we used specific scenarios for the system variants from Table 4.7. In these scenarios, the parameters used for the DPM system variants (power usage, transition rates etc) correspond to the real-world system [174, 188], while for the FX system variants we chose realistic values for the reliability, performance and cost of third-party services implementations. Second, to mitigate further the risk of accidentally choosing values that biased the EvoChecker evaluation, we defined a set of 30 different adaptation scenarios per FX system variant with varied services characteristics for each scenario.

4.3.1.3 Evaluation Methodology

We used the following MOGAs to evaluate the human-in-the-loop EvoChecker:

NSGA-II [57]: Given a population of individuals, NSGA-II establishes the fitness of each individual, then sorts each individual based on Pareto dominance and creates an offspring by means of mutation and crossover. It then ranks the combined

individuals from the population and offspring to generate sets of nondominated vectors as follows: all nondominated individuals that belong to the Pareto front are in rank 1, nondominated individuals after removing those individuals (in rank 1) comprise rank 2, and so on. Then for each rank, a crowding distance is computed for all the individuals in that rank. A new population is created using the best ranks, and if not all individuals from the same rank can be selected, the algorithm uses the crowding distance to select the most promising (diverse) individuals.

SPEA2 [221]: This is an archive-based algorithm that uses an external archive for storing nondominated individuals found during the search. The fitness of each individual is based on a *strength* metric, i.e., a combination of the number of individuals it dominates and the number of individuals it is dominated by, and a density estimation metric which indicates its distance from its k-th nearest neighbour. At the end of each generation, nondominated individuals are propagated into the archive. If the archive is full, SPEA2 uses an archive truncating operator to discard the individuals with the minimum distance to another individual (keeping boundary individuals). If the archive has available spaces, it is filled with the best dominated individuals (i.e., those with the minimum fitness) from the population. Next, a subset of individuals from the archive is selected, and after crossover and mutation, the generated offspring becomes the new population.

MOCell [166]: This is a cellular archive-based (like SPEA2) algorithm in which the population is structured in a multidimensional grid. The individuals form a set of overlapping neighbourhoods and an individual can cooperate only with members from its neighbourhood. Thus, for each individual an offspring is created by selecting two parents from its neighbourhood, and applying then crossover and mutation. That particular individual is replaced by its offspring if the former is Pareto dominated or if both are nondominated and the individual has lower crowding distance (in the neighbourhood) than the offspring. Nondominated individuals can enter the archive only if they have higher crowding distance than at least one individual from the archive. Finally, through a MOCell-specific feedback mechanism, a subset of individuals from the archive replace a corresponding subset of randomly selected individuals from the population. The updated population participates in the next generation.

In line with the standard practice for evaluating the performance of stochastic optimisation algorithms [10], we performed multiple (i.e., 30) independent runs for each system variant from Table 4.7 and each multiobjective optimisation algorithm, i.e.,

NSGA-II, SPEA2, MOCell and random search. Each run comprised 10,000 evaluations, each using a different initial population of 100 individuals, single-point crossover with probability $p_c = 0.9$, and single-point mutation with probability $p_m = 1/n_p$, where n_p is the number of configuration parameters for a particular system variant. All the experiments were run on a CentOS Linux 6.5 64bit server with two 2.6GHz Intel Xeon E5-2670 processors and 32GB of memory.

Obtaining the actual Pareto front for our system variants is unfeasible because of their very large configuration spaces. Therefore, we adopted the established practice [220] of comparing the Pareto front approximations produced by each algorithm with the *reference Pareto front* comprising the nondominated solutions from all the runs carried out for the analysed system variant. For this comparison, we employed the widely-used *Pareto-front quality indicators* below, and we will present their means and box plots as measures of central tendency and distribution, respectively:

I_ϵ (**Unary additive epsilon**) [223]. This is the minimum additive term by which the elements of the objective vectors from a Pareto front approximation must be adjusted in order to dominate the objective vectors from the reference front. This indicator presents convergence to the reference front and is *Pareto compliant*[5]. Smaller I_ϵ values denote better Pareto front approximations.

I_{HV} (**Hypervolume**) [222]. This indicator measures the volume in the objective space covered by a Pareto front approximation with respect to the reference front (or a reference point). It measures both convergence and diversity, and is strictly Pareto compliant [219]. Larger I_{HV} values denote better Pareto front approximations.

I_{IGD} (**Inverted Generational Distance**) [202]. This indicator gives an "error measure" as the Euclidean distance in the objective space between the reference front and the Pareto front approximation. I_{IGD} shows both diversity and convergence to the reference front. Smaller I_{IGD} values signify better Pareto front approximations.

We used inferential statistical tests to compare these quality indicators across the four algorithms [10, 117]. As is typical of multiobjective optimisation [220], the Shapiro-Wilk test showed that the quality indicators were not normally distributed, so we used the Kruskal-Wallis non-parametric test with 95% confidence level ($\alpha = 0.05$) to analyse the results without making assumptions about the distribution of the data or the homogeneity of its variance. We also performed a post-hoc analysis with pairwise comparisons

[5]Pareto compliant indicators do not "contradict" the order introduced by the Pareto dominance relation on Pareto front approximations [219].

between the four algorithms using Dunn's pairwise test, controlling the family-wise error rate with the Bonferroni correction $p_{crit} = \alpha/k$, where k is the number of comparisons.

4.3.1.4 Results and Discussion

RQ1 (Validation). We carried out the experiments described in the previous section and we report their results in Tables 4.9–4.11, and Figures 4.5–4.7. To allow for a fair comparison across the experiments comprising the 30 different FX adaptation scenarios, in Table 4.11 and Figure 4.7, and to avoid undesired scaling effects, we normalise the results obtained for each quality indicator per experiment within the range [0,1]. The '+' from the last column of the table entries indicate that the Kruskal-Wallis test showed significant difference among the four algorithms (p-value$<$0.05) for all six system variants and all Pareto-front quality indicators.

For both systems, human-in-the-loop EvoChecker with any MOGA achieved considerably better results than random search, for all quality indicators and system variants. The post hoc analysis of pairwise comparisons between random search and the MOGAs provided statistical evidence about the superiority of the MOGAs for all system variants and for all quality indicators. The best and, when obtained, the second best outcomes of this analysis per system variant and quality indicator are shaded and lightly shaded in the result tables, respectively. This superiority of the results obtained using EvoChecker with any of the MOGAs over those produced by random search can also be seen from the boxplots in Figures 4.5–4.7.

We qualitatively support our findings by showing in Figures 4.8 and 4.9 the Pareto front approximations achieved by EvoChecker with each of the MOGAs and by random search, for a typical run of the experiment for the DPM and FX system variants, respectively. We observe that irrespective of the MOGA, EvoChecker achieves Pareto front approximations with more, better spread and higher quality nondominated solutions than random search.

Considering all these results, we have strong empirical evidence that the human-in-the-loop EvoChecker significantly outperforms random search, for a range of system variants from two different domains, and across multiple widely-used MOGAs. This also confirms the challenging and well-formulated nature of the *multi-objective reconfiguration problem* we introduced in Section 4.1.3.

RQ2 (Comparison). To compare EvoChecker instances based on different MOGAs, we first observe in Tables 4.9–4.11 that NSGA-II and SPEA2 outperformed MOCell for all system variant–quality indicator combinations except DPM_Small (I_{IGD}). Between SPEA2 and NSGA-II, the former achieved slightly better results for the smaller configuration spaces of the DPM system variants (across all indicators) and for the I_{HV} indicator (across all system variants), whereas NSGA-II yielded Pareto-front approximations with better I_ϵ and I_{IGD} indicators for the larger configuration spaces of the FX system variants (except the combination FX_Small (I_ϵ)).

Additionally, we carried out the post-hoc analysis described in Section 4.3.1.3, for 9 system variants (counting separately the FX system variants with chosen services characteristics and those comprising the adaptation scenarios) × 3 quality indicators = 27 tests. Out of these tests, 22 tests (i.e., a percentage of 81.4%) showed high statistical significance in the differences between the performance achieved by EvoChecker with different MOGAs (Table 4.8). The five system variant–quality indicator combinations for which the tests were unsuccessful are: FX_Medium (I_ϵ), FX_Small_Adapt (I_ϵ), FX_ Medium_Adapt(I_ϵ), FX_Small(I_{IGD}) and FX_Medium(I_{IGD}).

These results show that, like for any well-formulated optimisation problem, different algorithms are more suitable in dealing with specific problems. They also confirm the generality of human-in-the-loop EvoChecker, showing that its functionality can be realised using multiple established MOGAs.

Table 4.8: System variants for which the MOGAs in rows are significantly better than the MOGAs in columns

		NSGA-II									SPEA2								MOCell								
		1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9
NSGA-II	I_ϵ															✓			✓		✓	✓						
	I_{HV}																			✓	✓	✓		✓	✓		✓	✓
	I_{IGD}																								✓			✓
SPEA2	I_ϵ	✓	✓	✓																✓	✓	✓	✓					
	I_{HV}				✓	✓	✓	✓	✓	✓										✓	✓	✓	✓	✓	✓	✓	✓	✓
	I_{IGD}	✓	✓	✓	✓																	✓			✓	✓	✓	✓
MOCell	I_ϵ																											
	I_{HV}																											
	I_{IGD}	✓	✓								✓																	

Key: 1:DPM_Small, 2:DPM_Medium, 3:DPM_Large, 4:FX_Small, 5:FX_Medium, 6:FX_Large, 7:FX_Small_Adapt, 8:FX_Medium_Adapt, 9:FX_Large_Adapt

Table 4.9: Mean quality indicator values for a specific scenario of the DPM system variants from Table 4.7

Variant	NSGA-II	SPEA2	MOCell	Random	
		I_ϵ **(Epsilon)**			
DPM_Small	0.0209	0.0130	0.0242	0.1403	+
DPM_Medium	0.0225	0.0123	0.0489	0.1996	+
DPM_Large	0.0229	0.0147	0.0884	0.2497	+
		I_{HV} **(Hypervolume)**			
DPM_Small	0.4455	0.4458	0.4396	0.4022	+
DPM_Medium	0.4487	0.4499	0.4386	0.3946	+
DPM_Large	0.4528	0.4549	0.4395	0.3947	+
		I_{IGD} **(Inverted Generational Distance)**			
DPM_Small	0.00023	0.00018	0.00016	0.00062	+
DPM_Medium	0.00024	0.00019	0.00028	0.00091	+
DPM_Large	0.00024	0.00020	0.00038	0.00109	+

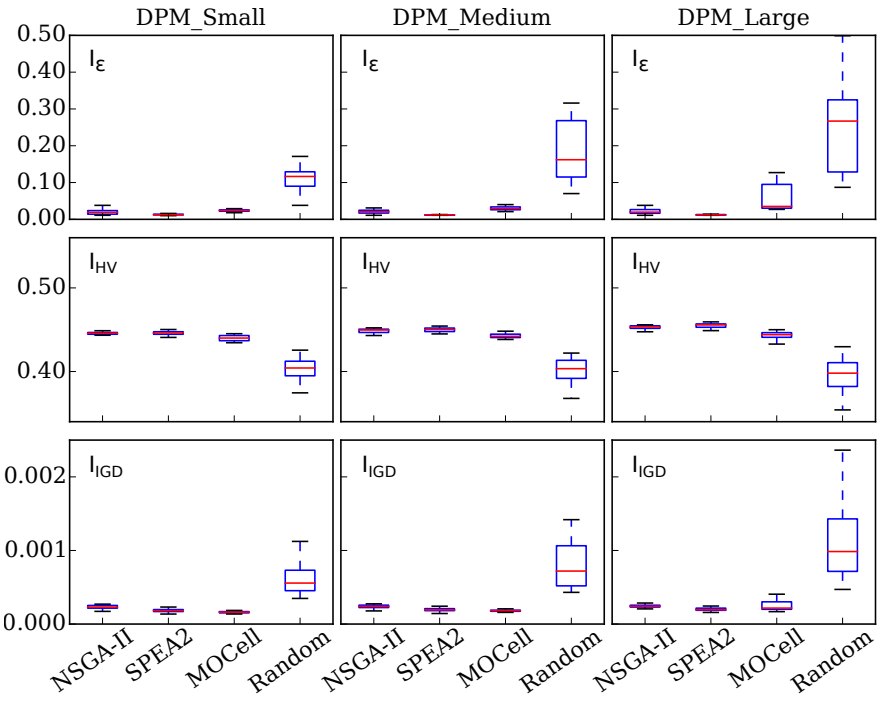


Figure 4.5: Boxplots for a specific scenario of the DPM system variants from Table 4.7, evaluated using the quality indicators I_ϵ, I_{HV} and I_{IGD}.

Table 4.10: Mean quality indicator values for a specific scenario of the FX system variants from Table 4.7

Variant	NSGA-II	SPEA2	MOCell	Random	
		I_ϵ **(Epsilon)**			
FX_Small	0.6258	0.5083	0.6745	2.2274	+
FX_Medium	1.6379	2.0105	2.0486	6.1529	+
FX_Large	3.8528	5.2777	4.6366	13.0234	+
		I_{HV} **(Hypervolume)**			
FX_Small	0.611	0.628	0.608	0.593	+
FX_Medium	0.719	0.725	0.702	0.606	+
FX_Large	0.657	0.675	0.633	0.555	+
		I_{IGD} **(Inverted Generational Distance)**			
FX_Small	0.00123	0.00129	0.00125	0.00145	+
FX_Medium	0.00192	0.00207	0.00200	0.00316	+
FX_Large	0.00244	0.00255	0.00272	0.00395	+

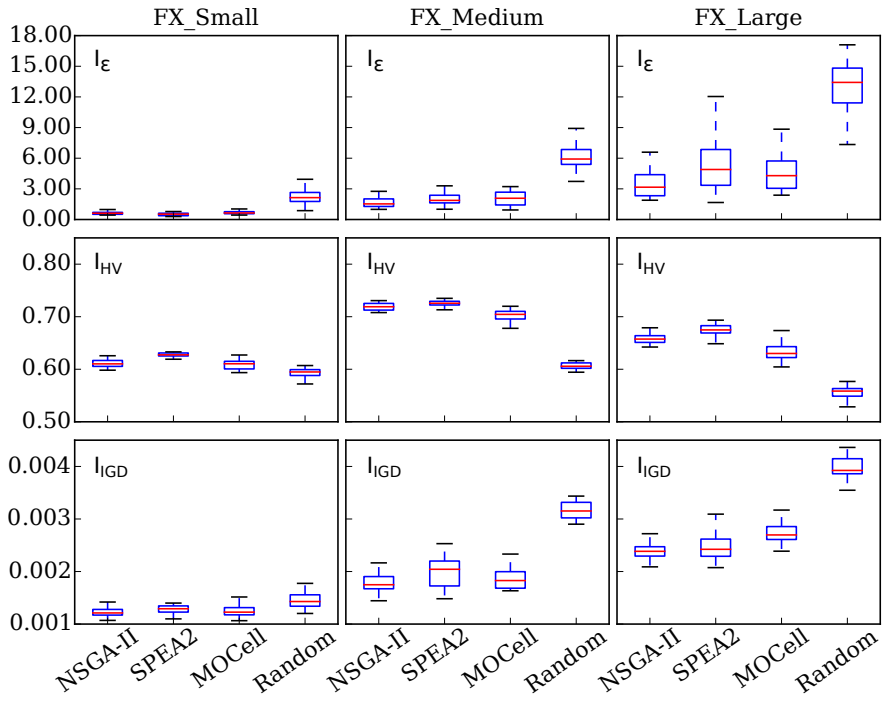


Figure 4.6: Boxplots for a specific scenario of the FX system variants from Table 4.7, evaluated using the quality indicators I_ϵ, I_{HV} and I_{IGD}.

Table 4.11: Mean quality indicator values across 30 different adaptation scenarios for the FX system variants from Table 4.7

Variant	NSGA-II	SPEA2	MOCell	Random	
			I_ϵ **(Epsilon)**		
FX_Small	0.2212	0.2209	0.2272	0.6200	+
FX_Medium	0.3393	0.3664	0.3645	0.7568	+
FX_Large	0.3396	0.3764	0.3625	0.7970	+
			I_{HV} **(Hypervolume)**		
FX_Small	0.9374	0.9914	0.9337	0.9016	+
FX_Medium	0.9514	0.9848	0.9219	0.8138	+
FX_Large	0.9467	0.9804	0.8962	0.7868	+
			I_{IGD} **(Inverted Generational Distance)**		
FX_Small	0.6365	0.5348	0.6390	0.8000	+
FX_Medium	0.5919	0.5790	0.6114	0.7957	+
FX_Large	0.5887	0.5622	0.6561	0.8884	+

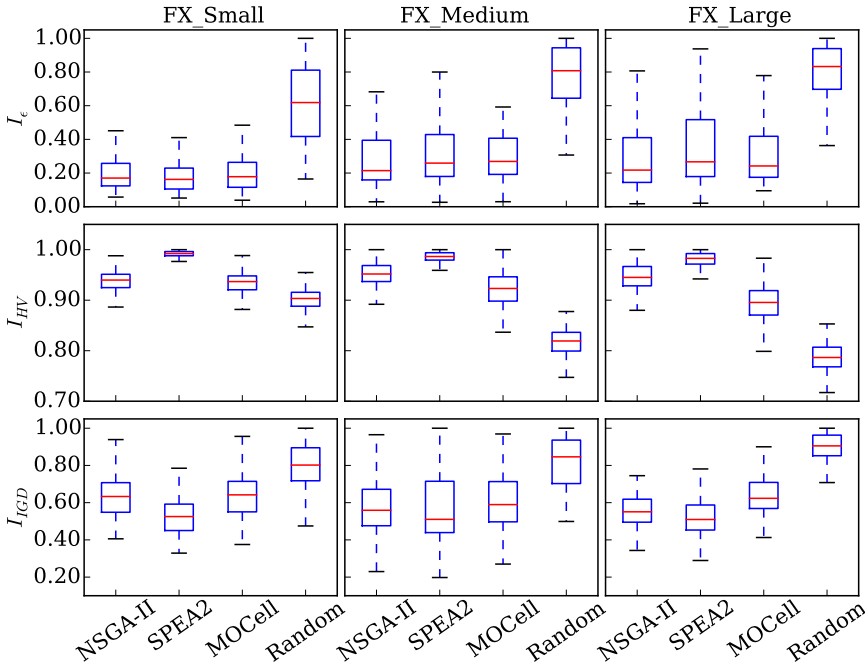
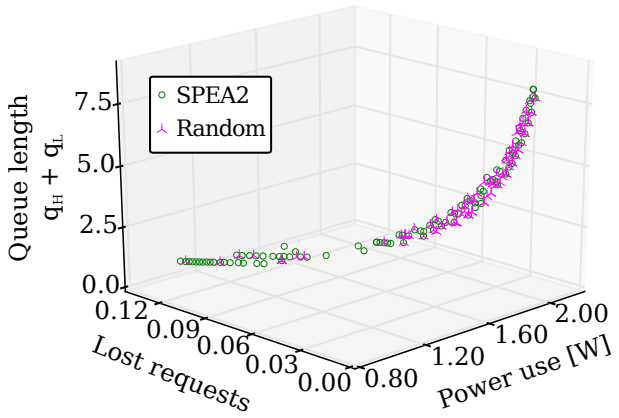
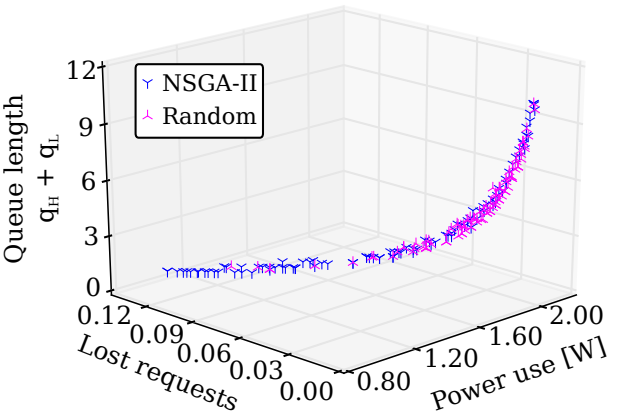


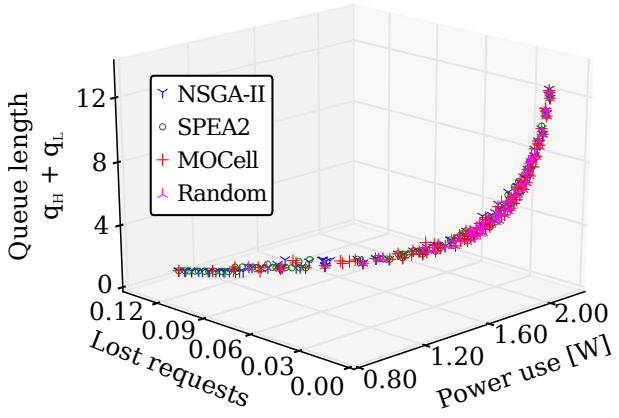
Figure 4.7: Boxplots for the FX system variants from Table 4.7 across 30 different adaptation scenarios, evaluated using the quality indicators I_ϵ, I_{HV} and I_{IGD}.



(a) DPM_Small

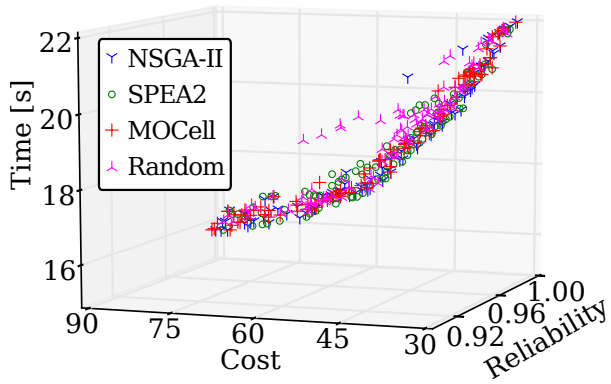


(b) DPM_Medium

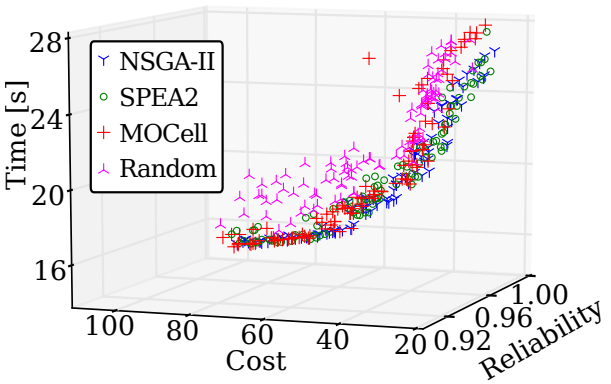


(c) DPM_Large

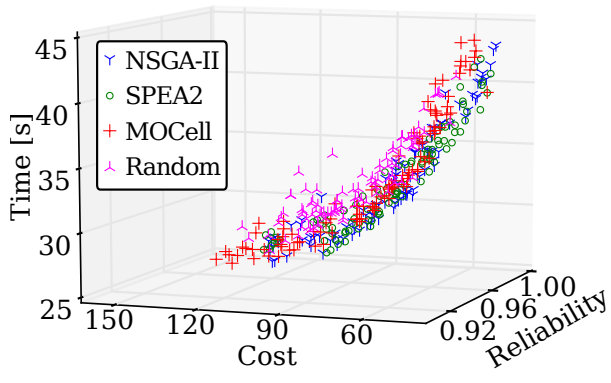
Figure 4.8: Typical Pareto front approximations for the DPM system variants and optimisation objectives R3–R5 from Table B.2.



(a) FX_Small



(b) FX_Medium



(c) FX_Large

Figure 4.9: Typical Pareto front approximations for the FX system variants and optimisation objectives R2–R4 from Table 4.4.

RQ3 (Insights). We performed qualitative analysis of the Pareto front approximations produced by EvoChecker, in order to identify actionable insights. We present this for the FX and DPM Pareto front approximations from Figures 4.8 and 4.9, respectively.

First, the EvoChecker results enable the identification of the "point of diminishing returns" for each system variant. The results from Figure 4.9 show that configurations with costs above approximately 52 for FX_Small, 61 for FX_Medium and 94 for FX_Large provide only marginal response time and reliability improvements over the best configurations achievable for these costs. Likewise, the results in Figure 4.8 show that DPM configurations with power use above 1.7W yield insignificant reductions in the number of lost requests, whereas configurations with even slightly lower power use lead to much higher request loss. This key information helps system experts to avoid unnecessarily expensive solutions.

Second, we note the high density of solutions in the areas with low reliability (below 0.95) for the FX system in Figure 4.9, and with high request loss (above 0.09) for the DPM system in Figure 4.8. For the FX system, for instance, these correspond to the use of the probabilistic invocation strategy, for which numerous service combinations can achieve similar reliability and response time with relatively low, comparable costs. Opting for a configuration from this area will make the FX system susceptible to failures, as when the only implementation invoked for an FX service fails, the entire workflow execution will also fail. In contrast, reliability values above 0.995 correspond to expensive configurations that use the sequential selection strategy; e.g., FX_Small must use the sequential strategy for the *Market watch* and *Fundamental analysis* services in order to achieve 0.996 reliability.

Third, the EvoChecker results reveal configuration parameters that QoS attributes are particularly sensitive to. For the FX system, for example, we noticed a strong dependency of the workflow reliability on the service invocation strategy and the number of implementations used for each service. Configurations from high-reliability areas of the Pareto front not only use the sequential strategy, but also require multiple services per FX service (e.g., three FX service providers are needed for success rates above 0.99).

Finally, we note EvoChecker's ability to produce solutions that: i) cover a wide range of values for the QoS attributes from the optimisation objectives of the FX and DPM systems; and ii) include alternatives with different trade-offs for fixed values of one of these attributes. Thus, for 0.99 reliability, the experiment from Figure 4.9 generated four alternative FX_Large configurations, each with a different cost and execution time. Similar observations can be made for a specific value of either of the other two QoS attributes. These results support the system experts in their decision making.

4.3.2 Automated EvoChecker Evaluation

4.3.2.1 Research Questions

We evaluated the automated EvoChecker to answer the research questions below.

RQ4 (Correctness): Can automated EvoChecker support dependable adaptation? With this research question we examine whether our approach can identify new effective configurations at runtime and if it can achieve this efficiently.

RQ5 (Validation): How does automated EvoChecker perform compared to random search? Following the standard practice in search-based software engineering [117], with this research question we aim to determine whether our approach "comfortably" outperforms random search.

RQ6 (Insights): How do instances of automated EvoChecker based on different archive updating strategies compare to each other? We used this research question to analyse the impact of various *archive updating strategies* in the performance of an EA. To this end, we study whether specific *strategies* improve the quality of an EA search and/or help identifying faster an effective configuration. We also investigate possible relationships between *archive updating strategies* and specific adaptation events.

4.3.2.2 Experimental Setup

For the experimental evaluation, we used two self-adaptive software systems from diverse application domains: i) the embedded UUV system from Section 2.2.1.1; and ii) the real-world foreign exchange (FX) service-based system from Section 4.1.

To evaluate the incremental EvoChecker for multiple configuration space sizes, we applied it to each of the system instances from Table 4.12. The column 'Details' shows for the UUV system the number of sensors, their measurement rates and the UUV speed, while for the FX system the number of third-party implementations for each service. The column 'Size' reports the size of the configuration space that an exhaustive search would need to explore using two-decimal precision for the real parameters and probability distributions of the probabilistic model template (cf. Table 4.5). Finally, the column 'T_{run}' shows the average time required by incremental EvoChecker to evaluate a configuration on a 2.6GhZ Intel Core i5 Macbook Pro computer with 16GB memory, running Mac OSX 10.9.

Table 4.12: Analysed system variants for the incremental EvoChecker

Variant	Details	Size	$\mathbf{T_{run}}[s]$
UUV_Medium	$n = 5, r_1, r_2, \ldots, r_5 \in [0Hz, 8Hz]$, $sp \in [0, 10m/s]$	1.04E+19	0.0076
UUV_Large	$n = 10, r_1, r_2, \ldots, r_{10} \in [0Hz, 8Hz]$, $sp \in [0, 10m/s]$	1.09E+35	0.1622
FX_Small	$n_1 = \cdots = n_4 = 3, n_5 = n_6 = 1$	8.49E+31	0.0312
FX_Medium	$n_1 = \cdots = n_6 = 4$	2.05E+65	0.0953

Moreover, we wanted to assess whether the use of an archive and a corresponding archive updating strategy has any effect on incremental EvoChecker. To this end, we performed a preliminary investigation aiming to identify several changes that cause each UUV and FX variant to adapt. These changes cover a wide range of the possible values that the observable parameters of each system variant can take (Table 4.13). Due to these changes, the systems experience problems while providing service (e.g., service degradation, violation of QoS requirements) and therefore are forced to adapt. Sensors in the UUV variants, beyond normal behaviour, encounter periods of unexpected changes (C1-C12) during which their rates change dramatically, including sensor failures and recovery from these failures, and significant variation in measurement rates. In FX, we define 13 changes (C1–C13) comprising sudden minor or significant increase in response time and decline in reliability of service implementations, and complete failure or recovery of service implementations.

4.3.2.3 Evaluation Methodology

Given that the optimisation of QoS objectives in the incremental EvoChecker is defined by a cost function (4.12), we opted for an elitist single objective GA. Recall that an elitist GA propagates the best individuals to the next generation. With elitism, if the GA discovers the best solution, then the entire population will eventually converge to this solution.

To investigate whether different archive updating strategies (cf. Def. 4.5) can improve the efficiency of incremental EvoChecker, we realised the strategies from (4.15)–(4.18). To this end, we created four different GA variants, each enhanced with one of the following archive updating strategies:

PGA: a *prohibitive strategy* (4.15) that does not keep any configurations in the archive. Thus, a search for a new configuration starts without using any prior knowledge.

Table 4.13: Changes in environment state of system variants used in automated EvoChecker

ID	UUV_Medium	UUV_Large	FX_Small	FX_Medium
C1	Nominal	Nominal	Nominal	Nominal
C2	Nominal	Nominal	Nominal	Nominal
C3	$r_1\downarrow$, $r_5\downarrow$	$r_1\downarrow$, $r_4\downarrow$ $r_9\downarrow$	$r_{11}\downarrow$, $r_{13}\downarrow$	$r_{11}\downarrow$, $r_{13}\downarrow$, $r_{14}\downarrow$
C4	$r_1\leftrightarrow$, $r_5\leftrightarrow$	$r_1\leftrightarrow$, $r_4\leftrightarrow$ $r_9\leftrightarrow$	$r_{11}\leftrightarrow$, $r_{13}\leftrightarrow$	$r_{11}\leftrightarrow$, $r_{13}\leftrightarrow$, $r_{14}\leftrightarrow$
C5	$r_2\downarrow$, $r_4\downarrow$	$r_2\downarrow$, $r_4\downarrow$, $r_8\downarrow$, $r_{10}\downarrow$	$r_{21}\downarrow$, $r_{22}\downarrow$	$r_{21}\downarrow$, $r_{22}\downarrow$, $r_{24}\downarrow$
C6	$r_2\leftrightarrow$, $r_4\leftrightarrow$	$r_2\leftrightarrow$, $r_4\leftrightarrow$, $r_8\leftrightarrow$, $r_{10}\leftrightarrow$	$r_{21}\leftrightarrow$, $r_{22}\leftrightarrow$	$r_{21}\leftrightarrow$, $r_{22}\leftrightarrow$, $r_{24}\leftrightarrow$
C7	$r_2\downarrow$	$r_8\downarrow$, $r_{10}\downarrow$	$r_{11}\downarrow$, $r_{13}\downarrow$	$r_{11}\downarrow$, $r_{13}\downarrow$, $r_{14}\downarrow$
C8	$r_2\leftrightarrow$	$r_8\leftrightarrow$, $r_{10}\leftrightarrow$	$r_{11}\leftrightarrow$, $r_{13}\leftrightarrow$	$r_{11}\leftrightarrow$, $r_{13}\leftrightarrow$, $r_{14}\leftrightarrow$
C9	$r_1\downarrow$, $r_5\downarrow$	$r_1\downarrow$, $r_5\downarrow$ $r_9\downarrow$	$t_{41}\uparrow$, $t_{42}\uparrow$	$t_{41}\uparrow$, $t_{42}\uparrow$, $t_{44}\uparrow$
C10	$r_1\leftrightarrow$, $r_5\leftrightarrow$	$r_1\leftrightarrow$, $r_5\leftrightarrow$ $r_9\leftrightarrow$	$t_{41}\leftrightarrow$, $t_{42}\leftrightarrow$	$t_{41}\leftrightarrow$, $t_{42}\leftrightarrow$, $t_{44}\leftrightarrow$
C11	$r_1\downarrow$, $r_3\downarrow$, $r_5\downarrow$	$r_1\downarrow$, $r_3\downarrow$, $r_5\downarrow$ $r_7\downarrow$, $r_9\downarrow$, $r_{10}\downarrow$	$t_{51}\uparrow$, $t_{52}\uparrow$	$t_{51}\uparrow$, $t_{52}\uparrow$, $t_{53}\uparrow$
C12	$r_1\leftrightarrow$, $r_3\leftrightarrow$, $r_5\leftrightarrow$	$r_1\leftrightarrow$, $r_3\leftrightarrow$, $r_5\leftrightarrow$ $r_7\leftrightarrow$, $r_9\leftrightarrow$, $r_{10}\leftrightarrow$	$t_{51}\leftrightarrow$, $t_{52}\leftrightarrow$	$t_{51}\leftrightarrow$, $t_{52}\leftrightarrow$, $t_{53}\leftrightarrow$
C13			$r_{11}\downarrow$, $r_{12}\downarrow$, $r_{21}\downarrow$, $r_{22}\downarrow$, $r_{31}\downarrow$, $r_{33}\downarrow$, $r_{42}\downarrow$, $r_{43}\downarrow$	$r_{11}\downarrow$, $r_{12}\downarrow$, $r_{13}\downarrow$, $r_{21}\downarrow$, $r_{22}\downarrow$, $r_{31}\downarrow$, $r_{33}\downarrow$, $r_{43}\downarrow$, $r_{44}\downarrow$ $r_{51}\downarrow$, $r_{52}\downarrow$, $r_{54}\downarrow$ $r_{62}\downarrow$, $r_{64}\downarrow$

Key
$\downarrow$: change (decrease) in environment characteristic
$\uparrow$: change (increase) in environment characteristic
$\leftrightarrow$: recovery of environment characteristic

CRGA: a *complete recent strategy* (4.16) that puts in the archive the entire population from the current adaptation step and discards all previous configurations.

LRGA: a *limited recent strategy* (4.17) that stores in the archive the two best configurations (i.e., $x = 2$) from the current adaptation step, and removes all the other configurations from the archive.

LDGA: a *limited deep strategy* (4.18)that accumulates in the archive the two best configurations (i.e., $x = 2$) from *all* previous adaptation steps. If the archive size exceeds the initial size of the GA population, then a random selection is carried out to select the configurations that will comprise the seed for the next search.

We adopted the established procedure in search-based software engineering for the analysis of optimisation algorithms [10]. Thus, for all system variants from Table 4.12 we carried out 30 independent runs for each adaptation event per optimisation algorithm. All algorithms used a population of 50 individuals. The GAs used single-point crossover with probability $p_c = 0.9$ and single-point mutation with probability $p_m = 1/n_k$, where n_k is the number of system configuration parameters from the configuration space *Cfg*. Each algorithm was executed for 5000 iterations. After normalisation, we assigned the maximum cost of 1.00 for each event in which an algorithm failed to find a configuration satisfying QoS constraints. When no improvement was detected for 1000 successive iterations (i.e., 20% of the allocated evolution time), the evolution terminated early. The solution corresponding to the best individual from the last population was used to reconfigure the system. We use this configuration to compare the performance of the optimisation algorithms and answer research questions **RQ4**–**RQ6**.

Following the standard advice for assessing the performance of optimisation algorithms, we used inferential statistical tests [10, 50]. First, we analysed the normality of data and confirmed its deviation from the normal distribution using the Shapiro-Wilk test. Then, we used the non-parametric tests Mann–Whitney and Kruskal-Wallis with 95% confidence level ($\alpha = 0.05$) to analyse the results without making assumptions about the data distribution or the homogeneity of its variance. Also, to compare the GA variants, we ran a post-hoc analysis using Dunn's pairwise test, controlling the family-wise error rate using the Bonferroni correction $p_{crit} = \alpha/k$, where k is the number of comparisons.

Finally, when statistical significance exists, we establish the practical importance of the observed effect. Therefore, we used the Varga and Delaney's effect size measure [10, 203]. When comparing algorithms A and B, this measure returns the probability $A_{AB} \in [0, 1]$ that algorithm A will yield better results than algorithm B. For instance, if $A_{AB} = 0.5$ then the algorithms are equivalent, while if $A_{AB} = 0.8$ then algorithm A will achieve better results 80% of the time.

4.3.2.4 Results and Discussion

RQ4 (Correctness). We begin the presentation of our results by examining whether our approach can identify new effective configurations in response to unexpected environment and/or system events. To answer this research question we performed two types of experiments. First, we assessed the effectiveness of the selected configurations compared to those generated by exhaustive search. To make the configuration

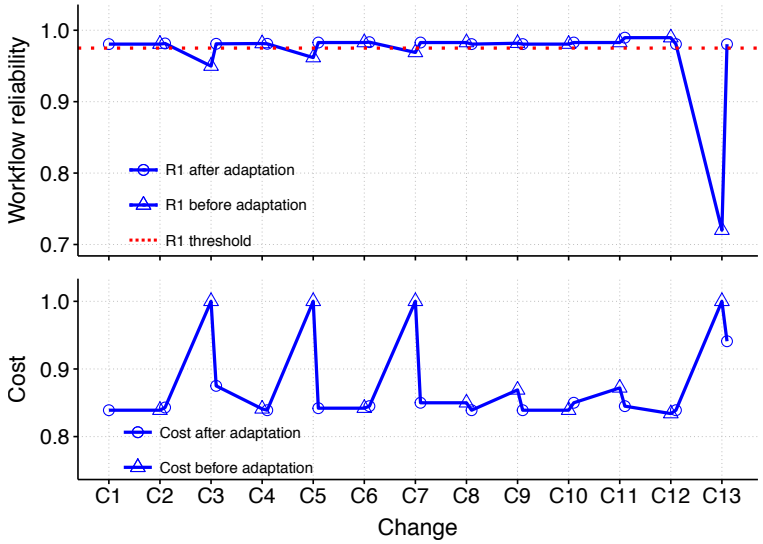


Figure 4.10: Variation in workflow reliability and system cost of the FX_Small variant due to the changes from Table 4.13 and system adaptation using the incremental EvoChecker with no archive use (i.e., PGA).

space size tractable for exhaustive search, we used the smallest system variant, i.e., UUV_Medium, and disabled three of its sensors, leaving less than 2.56E+9 possible configurations. We also disregarded the adaptation time, since it is too large for exhaustive search. For the same reason, we performed this assessment on a subset of the UUV changes (i.e., C1, C3, C4); these changes correspond to a representative sample of the UUV changes from Table 4.13. For all the events, our approach found configurations satisfying system QoS requirements with cost less than 9% of the optimal configuration reported by exhaustive search. Both time and memory overheads incurred by exhaustive search were approximately two orders of magnitude larger than our approach.

For the second experiment, we analysed how the adverse events in FX_Small system from Table 4.13 affected its compliance with QoS requirement R1 (i.e., workflow reliability) and varied system cost before (using the current configuration) and after (using the new configuration) each adaptation. Figure 4.10 depicts a typical run (timeline) of these changes and the impact of the configurations selected by the no-archive version of incremental EvoChecker (i.e., PGA) in workflow reliability and system cost.

First, irrespective of the change in environment state, either being a serious decrease in workflow reliability or a moderate increase in response time, the system always managed to successfully self-adapt. To this end, our approach always identified configurations that met requirement R1 and maintained a balanced system cost of approxi-

mately 0.845. Given that searching exhaustively the configuration space is unfeasible and that the average running time for evaluating a particular configuration is less than 1s (cf. Table 4.12), these experimental results indicate that our approach can support system adaptation.

We also analysed changes C3, C5, C7 and C13, in which the system exhibited significant decrease in workflow reliability, caused by decrease in reliability of the service implementations used at various points in time. Due to this abrupt change, the currently used service implementations failed to meet requirement R1 and the incremental EvoChecker was invoked to carry out the search for a new configuration. As an example, for change C13, the system experienced a serious disruption in about 50% of the available service implementations. As a result, workflow reliability fell to only 72%. The newly found configuration restored compliance with R1 (i.e., approximately 98.5%), but increased the probabilities of using more expensive implementations, yielding a significantly higher expected system cost of 0.935.

Another interesting observation concerns change C10 (cf. Table 4.13) in which two previously under-performing service implementations (those with increased response time t_{41} and t_{42}) recover. Although no requirement violation occurs, i.e., workflow reliability R1 is not affected by this change, the system cost corresponding to the new configuration selected by EvoChecker is slightly higher compared to the configuration before the change. Since for each change PGA starts a new search and does not use any knowledge gained from previous adaptation steps, this is expected. As we explain in **RQ6**, this issue can be addressed using one of the other archive updating strategies which seed a new GA search with configurations from the archive.

RQ5 (Validation). To answer this research question we compared the no-archive version of incremental EvoChecker (i.e., PGA) with random search (RS). In order to be concise, we include a representative sample of reconfiguration events. Thus, Figures 4.11 and 4.12 show the evolution of the algorithms every 500 iterations for the FX_Small variant for changes C4, C7, C11 and C13, and for the UUV_Large variant for changes C7 and C12, respectively. When an algorithm terminated early, we propagated the last cost to the remaining evolution stages (i.e., until the 5000th iteration). An asterisk $*$ next to each algorithm's boxplot denotes when the algorithm terminated for *all* 30 runs.

For both variants of the FX and UUV systems and for all 25 events, the incremental EvoChecker employing PGA identified configurations that met QoS requirements and achieved better cost than RS. We obtained statistical significance (p-value <0.05) using the Mann-Whitney test for all system variants and for all events, with the p-value being

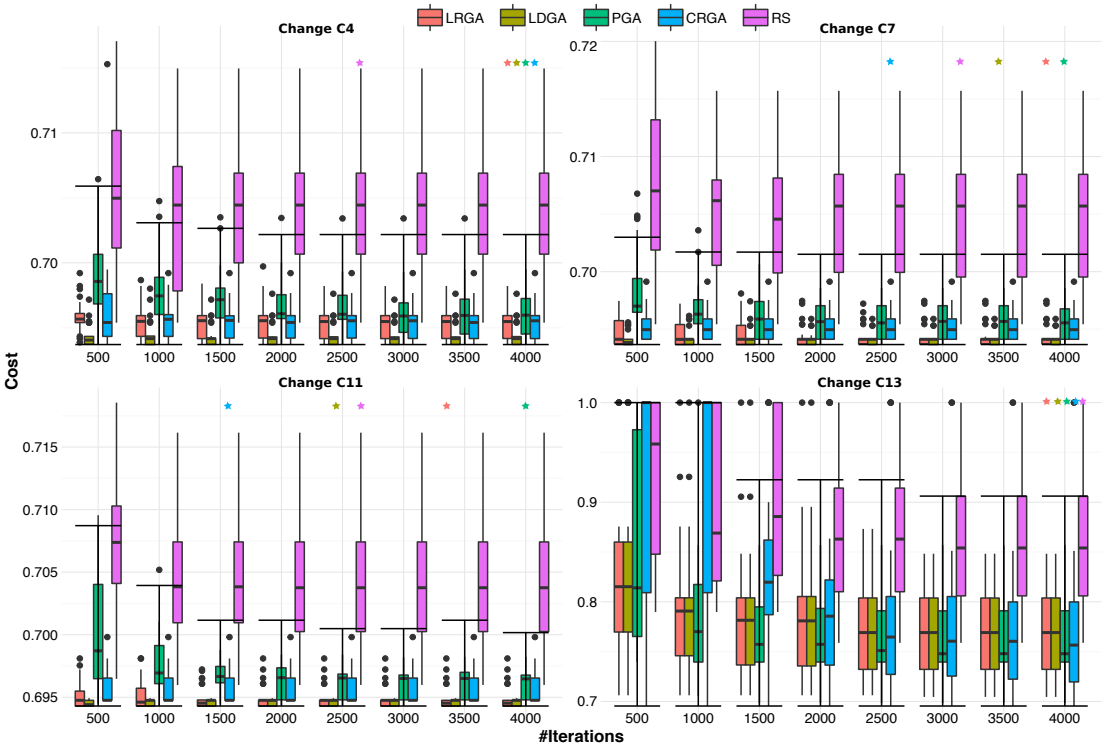


Figure 4.11: Boxplots for changes in environment state C4, C7, C11, C13 of the FX_Small system variant using LRGA, LDGA, PGA, CRGA, and RS. The asterisk next to each algorithm's boxplot signifies when the algorithm terminated for *all* 30 runs.

in the range [1.689E-02, 1.669E-11]. In fact, as the size of the system increases, PGA's ability to outperform RS becomes more evident.

We also measured the improvement magnitude using the $A_{\text{PGA,RS}}$ effect size metric [203]. For all evaluated events and evolution stages, the effect size was large with $A_{\text{PGA,RS}} \in [0.696, 1.00]$. Thus, PGA achieved better results than RS at least 69.6% of the time, while in some events, especially for the larger system variants FX_Medium and UUV_Large, the dominance reached 100%.

Another interesting finding concerns the evolution of the populations of these algorithms. Despite the overall performance difference, at the beginning of the evolution, i.e., 200-300 iterations, both the p-value and effect size are on the lower end of their respective value ranges. During these iterations, PGA operates pseudo-randomly and the impact of its selection and reproduction mechanisms, i.e., crossover and mutation, are not strong yet. As the evolution progresses, the performance gap between PGA and RS increases, reaching eventually the upper end of the p-value and effect size ranges.

Considering these results, we conclude that our GA-based approach using a prohibitive selection strategy (PGA) significantly outperforms random search (RS) with

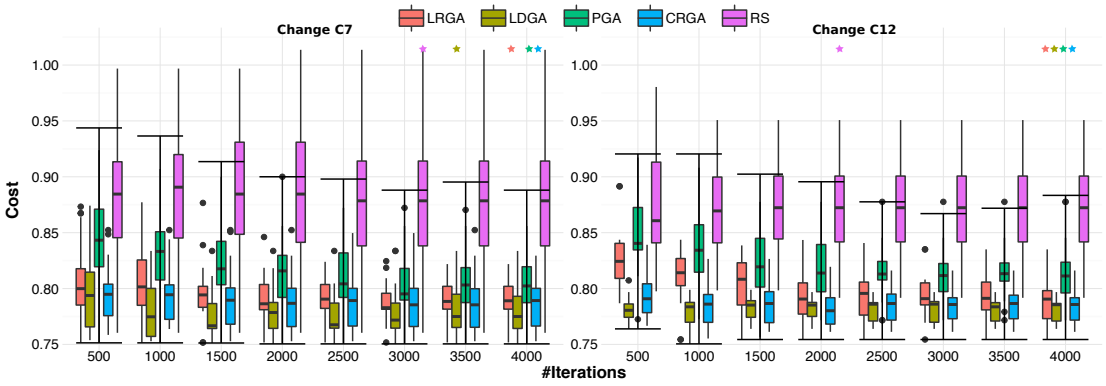


Figure 4.12: Boxplots for changes C7, C12 of the UUV_Large system variant using LRGA, LDGA, PGA, CRGA, and RS. The asterisk next to each algorithm's boxplot signifies when the algorithm terminated for *all* 30 runs.

large effect size in all adaptation steps and for all FX and UUV system variants. Thus, the use of evolutionary search-based approaches produces better quality configurations.

RQ6 (Insights). We analysed the system configurations selected by a GA using the archive updating strategies: prohibitive (PGA), complete recent (CRGA), limited recent (LRGA) and limited deep (LDGA) in order to identify actionable insights. Note that these strategies are used on top of a basic GA and thus have similar overheads (i.e., negligible CPU and memory use). Hence, the incurred overheads from the use of these strategies are not discussed further. In the interest of conciseness we show a subset of these adaptation steps; similar reasoning applies for the other steps. Table 4.14 shows an excerpt of the pairwise comparisons carried out to check for significant difference and, when the difference exists, its effect size in parenthesis.

First, for change C1 (not shown in Table 4.14) and for all FX and UUV variants, all examined archive updating strategies identified configurations of comparable quality. No statistical difference was detected in any evolution stage for this event. Since all algorithms used a randomly generated initial population for change C1, this observation was not surprising.

Second, we found that GA variants using the archive (LRGA, CRGA, LDGA) performed significantly better than PGA for changes C2–C12 in FX and for most events in UUV during the majority of the evolution stages. No comparison showed statistical significance in favour of PGA for any change or evolution stage. As expected, as the evolution progressed all the GA variants had the opportunity to refine their solutions and the performance gap between the algorithms decreased. More specifically, there

Table 4.14: Pairwise comparison of archive selection strategies for various stages of changes C4 and C11 of the FX variants showing the significantly better strategy and effect size (in parenthesis); Key: S=Small, M=Medium, L=Large

	C4				C11			
Strategies	1000	2000	3000	4000	1000	2000	3000	4000
				FX_Small				
RS vs PGA	PGA(L)	PGA(L)	PGA(L)	PGA(L)	PGA(L)	PGA(L)	PGA(L)	PGA(L)
PGA vs LRGA	LRGA(L)	LRGA(M)	LRGA(M)	LRGA(S)	LRGA(L)	LRGA(L)	LRGA(L)	LRGA(M)
PGA vs CRGA	CRGA(L)	CRGA(S)	—	—	—	—	—	—
LRGA vs CRGA	—	—	—	—	LRGA(L)	LRGA(L)	LRGA(L)	LRGA(L)
				FX_Medium				
RS vs PGA	PGA(L)	PGA(L)	PGA(L)	PGA(L)	PGA(L)	PGA(L)	PGA(L)	PGA(L)
PGA vs LRGA	LRGA(L)	LRGA(M)	LRGA(S)	—	LRGA(L)	LRGA(M)	LRGA(S)	—
PGA vs CRGA	CRGA(M)	CRGA(S)	—	—	—	—	—	—
LRGA vs CRGA	LRGA(S)	LRGA(S)	LRGA(S)	LRGA(S)	—	—	—	—

was a distinct performance gap favouring LRGA, CRGA and LDGA at the early evolution stages (p-value $\in$ [3.38E-5,1.67E-11]), while PGA was able to find configurations that achieve similar cost towards the end of evolution (p-value $\approxeq$ 0.05 in some cases). Looking at change C4 in Figure 4.11, for instance, PGA is significantly worse until the 2500th iteration, but it approaches the others after that.

Third, in changes with statistical difference between PGA and the other variants, we observed a similar declining trend regarding the effect size. At the beginning of the evolution, the effect size is mostly large ([0.69.0.88] and [0.77,1.0] for UUV and FX, respectively), at the intermediate stages it changes to medium/small before it becomes small/negligible towards the end. Given these observations, we can state that using an archive updating strategy to select configurations from the archive and seed the initial population produces better configurations and faster, compared to a prohibitive strategy that ignores the archive. Given sufficient time, however, PGA will potentially catch up. Thus, archive-based GA variants are useful in the frequently encountered situations where the reconfiguration time and/or computation resources are limited.

Fourth, the archive-based GA variants (LRGA, CRGA, LDGA) identified configurations of similar quality to each other, demonstrating effective use of the archive. The post-hoc analysis, however, showed a performance difference between the three variants. In particular, we obtained statistically significant results in favour of LDGA against LRGA in 202 out of 500 tests (40.4%). For most changes, this difference concerned the first few evolution stages; after that LRGA performed similarly (e.g., C7 and C11 in Figure 4.11). Furthermore, CRGA failed to produce better configurations than LDGA for any change and system variant, whereas it was marginally better than LRGA

(6.2%) in changes that had similar characteristics to the preceding change. Like before, the performance difference involved only the initial stages. On the other hand, both LRGA and LDGA outperformed CRGA in a range of changes and evolution stages. We obtained statistical difference favouring LRGA and LDGA in 18.2% (91/500) and 47.8% (239/500) of these tests, respectively. This difference occurs because CRGA's population already identified good configurations and/or converged to a particular area in fitness landscape. Since population variation is achieved only through crossover and mutation, CRGA finds difficulties to evolve the population in successive generations and produce better configurations. This leads to stagnation and early termination; see for instance changes C7 and C11 in Figure 4.11 in which CRGA terminated in the 2500th and 1500th iteration, respectively. Therefore, reusing the final population from the current adaptation event does not offer a distinct advantage in producing better configurations over the other strategies. However, exploiting a subset of configurations from previous reconfiguration events (e.g, LDGA) could speed up the search significantly.

Finally, we note the inability of any variant to deal efficiently with disruptive change C13 affecting FX. For this event, about 50% of the available service implementations suffered a serious service degradation (cf. Table 4.13). For C13, we did not find any statistical significance between PGA, CRGA, LRGA and LDGA in any evolution stage in both FX system variants (Figure 4.11). Moreover, at the early evolution stages, CRGA had difficulties to select configurations that satisfy QoS requirements; its cost is close to the maximum value. Hence, when a disruptive change occurs, it does not have much impact which archive updating strategy is used. Using instead a population that is not biased towards a particular area (due to previous experience) would facilitate landscape exploration.

We suppose that a hybrid approach which considers the types of changes in the system and its environment would be more effective. In this hybrid approach, some of the initial population would be derived from the archive (to exploit knowledge gained from previous reconfiguration events) and some would be randomly generated (to enable exploration of new events). The ratio between exploration and exploitation should be based on the expected ratio between small changes and radical changes in the environment. We discuss this idea further, as part of future work, in Section 7.2.

4.3.3 Threats to Validity

Several *construct*, *internal*, and *external* validity threats could affect the validity of the experiments conducted in this chapter.

Construct validity threats correspond to the methodology adopted when designing the experimental study and any underpinning assumptions. This includes any assumptions and simplifications made when modelling the DPM, FX and UUV systems. To mitigate this threat, the DPM system, model and requirements are based on a validated real-world case study taken from the literature [174, 188], which we are familiar with from our previous work [38]. For the FX system, the model and requirements were developed in close collaboration with a foreign exchange domain expert, while for the UUV system, these are based on the specification of a real sensor (and also used in our work in Chapter 3). Also, the environment changes cover a wide range of system scenarios that could cause service degradation and/or violation of QoS requirements, including minor changes and disruptive events.

Internal validity threats might be due to any bias introduced when establishing the causality between the adaptation steps and the evolutionary algorithms employed in our study. To mitigate this threat, we followed the established practice in search-based software engineering [10, 117]. In particular, we reported results over 30 independent runs of each experiment and used inferential statistical tests to check for significant difference in the performance of the algorithms. To this end, we evaluated whether the data conformed to the normal distribution using the Shapiro-Wilk test and used the non-parametric tests Mann-Whitney and Kruskal-Wallis to check for statistical significance. We also conducted a post-hoc analysis using Dunn's pairwise test. All these tests used a 95% confidence level; hence, the probability of committing a Type I error is 0.05, which is the recommended value in empirical studies in this area. Finally, we employed the Varga and Delaney's effect size measure to establish the magnitude of an improvement.

External validity threats might be due to the difficulty of representing a self-adaptive software system and its QoS requirements as a reconfiguration problem using EvoChecker constructs (4.4)–(4.6), constraints (4.8) and optimisation objectives (4.9) or cost (4.12). We limit this threat by specifying EvoChecker probabilistic model templates in an extended version of the high-level modelling language of PRISM [149], a widely-used probabilistic model checker. Moreover, given the generality of the EvoChecker constructs (4.4)–(4.6), other probabilistic modelling languages (e.g., those of the model checkers MRMC [132, 133] and Ymer [214]) can be naturally supported. Additionally, EvoChecker supports a wide range of probabilistic models and temporal logics (Table 4.2). We also examined various archive updating strategies, but other more sophisticated strategies can be developed. Finally, to further reduce the risk that EvoChecker might be difficult to use in practice, we validated it through application to several variants of three realistic software systems with diverse characteristics in terms of ap-

plication domain, size, complexity and requirements. Nevertheless, we are aware that our findings are by no means conclusive for all types of software systems, and more experiments are needed to confirm the generality of the EvoChecker approach and tool.

4.4 Related Work

Search-Based Software Engineering (SBSE). Search-based techniques [117] have been successfully used in areas ranging from project management [74, 178, 194] and testing [9, 86, 114] to effort estimation [160], software repair and evolution [42, 173] and software product lines [113, 185]. However, as reported in Harman *et al.*'s recent SBSE survey [116], this success does not yet extend to model checking and the existing research focuses on design-time activities. In [129, 134], genetic evolution is applied to synthesise model checking specifications, while in [5, 6] ant colony optimisation is used for generating counterexamples in large stochastic models.

While there is increasing interest on dynamic adaptive search-based techniques [112], their use in reconfiguring software systems based on QoS requirements is rather limited. Harman et al. [115] report that a combination of machine learning and search-based techniques will enable software systems to adapt while providing service. Early work in this direction is presented in [51]. The only other approach that we are aware of in this area is Plato [175], which employs genetic algorithms in the decision-making process of a self-adaptive system and generates new configurations that balance functional and non-functional requirements. However, Plato does not consider environment or system stochasticity, as EvoChecker does with its probabilistic model template, nor it uses any knowledge acquired during system operation to speed up the search, as incremental EvoChecker does with its archive and archive updating strategies.

Our work is also related to research that explores ways to incorporate problem specific knowledge into an evolutionary algorithm through seeding its initial population [103]. If prior knowledge is available or can be generated with reasonable computational effort, effective seeding may yield better quality solutions and lead to faster convergence [135]. The effect of various seeding options (between 25%-100% of the population size) was studied in [168] for the travelling salesman and the job-shop scheduling problems. The authors reported that seeding produced most of the time significantly better solutions than no seeding, although a 100% seed did not always generate better results. In the domain of search-based software testing, Fraser and Arcuri [85] assessed the effectiveness of various seeding strategies for generating test cases in object-oriented languages. They found that the impact of effective seeding is heavier during the early

stages of the search, while weaker seeding strategies or no seeding will perform similarly from the intermediate stages onwards. These observations are in line with our findings regarding the impact of the archive updating strategies (4.15)–(4.18).

Stochastic Controller synthesis. EvoChecker also partially overlaps with research carried out in the area of stochastic controller synthesis, in which formally verified stochastic controllers are used to disable certain (controllable) system behaviours or to vary the probability with which these behaviours occur.

Draeger et al. [65] propose the synthesis of a multi-strategy controller that enables a set of actions at any state and which is optimally permissive with respect to a penalty function. Irrespective of the action carried out, the controller guarantees compliance with system requirements. However, unlike our work which covers the full PCTL and CSL, [65] focuses only on probabilistic reachability and expected total rewards.

Moreno et al. [162] propose a controller synthesis approach by combining lookahead and latency awareness. Lookahead projects the expected system evolution over a limited horizon, while latency awareness considers the time between making and realising an adaptation decision. The synthesised controller performs a limited lookahead, but it ignores any previous knowledge and thus fails to support incremental synthesis.

A complementary approach to incremental controller synthesis is proposed by Ulusoy et al. [201]. The key idea is based on partitioning the synthesis task into several steps and refine the controller incrementally. Initially the technique considers a high-level system model and adds extra details as the synthesis progresses, until a termination criterion is met (e.g., exhausted computational resources). Unlike EvoChecker, though, which supports a variety of specification logics (cf. Table 4.2), this work supports only specifications defined in linear temporal logic.

Runtime Quantitative Verification. Our work is also related to recent advances in runtime quantitative verification [32]. These advances include *compositional*, *incremental* and *parametric* verification, and are discussed in detail in Section 2.2.3. We also presented in Chapter 3 how caching, limited lookahead and nearly-optimal reconfiguration can improve RQV efficiency. Each of these variants reduces the computation and memory overheads of RQV, but their applicability is limited to particular adaptation problems and specific types of Markov models and properties. EvoChecker, on the other hand, is model and property agnostic. Finally, in Chapter 5, we will present an approach that extends the applicability of RQV to distributed self-adaptive software systems.

4.5 Summary

In this chapter we introduced EvoChecker, a tool-supported search-based approach that improves the efficiency of runtime quantitative verification, especially in systems with large configuration space sizes. Given as input a probabilistic model template that encodes configuration parameters and a set of QoS requirements specifying constraints and objectives (or a cost) to be optimised, EvoChecker employs evolutionary algorithms to find effective configuration(s) and to drive adaptation. We developed two EvoChecker variants. The former, human-in-the-loop EvoChecker, uses multi-objective evolutionary algorithms to generate the Pareto-optimal configurations and then requests from a system expert to validate adaptation decisions (or select new configurations). The other, incremental EvoChecker, uses single objective evolutionary algorithms, maintains an archive of configurations from recent adaptations, and uses this archive to seed the initial population of an evolutionary algorithm before a new search. We also defined several specialised archive updating strategies and developed prototype implementations.

We evaluated each EvoChecker variant within two case studies from different application domains, showing its effectiveness, applicability and flexibility. Our results indicate that both EvoChecker variants reduce significantly the RQV overheads for reconfiguring a self-adaptive system. Human-in-the-loop EvoChecker can generate Pareto-optimal approximation sets and help system experts to make informed decisions (e.g., identify "point of diminishing returns", find configuration parameters that affect QoS attributes more). We also found that NSGA-II and SPEA2 performed equally good in both case studies and for all analysed quality indicators. Hence, any of these algorithms is a good choice for instantiating the human-in-the-loop EvoChecker. In the incremental EvoChecker, strategies that make use of the archive can identify effective configurations much faster than strategies that do not use the archive. Thus, storing configurations from recent adaptations in an archive and using this archive to seed a new population can speed up the search, especially if similar environment states are encountered often.

We should note that selecting between the EvoChecker variants depends on the characteristics of each self-adaptive system including the number of QoS requirements, the time available for adaptation and whether adaptation decisions must be validated by a human expert. Finally, the EvoChecker approach (especially the human-in-the-loop variant) can be also used at design-time for the engineering of software systems (not necessarily self-adaptive). Thus, EvoChecker can be employed to generate probabilistic models that meet the QoS requirements of a software system, and then human experts can select a suitable model and use it as a basis for the system implementation.

Chapter 5

Extending RQV With Decentralised Control Loops

Since its introduction in [32, 38, 68], RQV has attracted the interest of many researchers from the area of self-adaptive software. Most of their research efforts explore how to reduce verification overheads and extend the applicability of the technique to larger and more complex self-adaptive systems. Illustrative examples include the work reviewed in Section 2.2.3 and our efficient RQV techniques based on conventional software engineering and search-based approaches, introduced in Chapters 3 and 4, respectively.

Despite the advances made by this recent research, the proposed RQV variants have been used to develop *centralised-control* self-adaptive software; see Section 5.4 for a discussion of related work. This is feasible only for self-adaptive systems whose stochastic models are small enough to be analysed fast and with acceptable overheads. Also, the use of centralised control in distributed systems (e.g., service-based and multi-agent applications) introduces a single point of failure. If the control component fails, the entire system will completely lose the ability to adapt to changes and may fail too.

In this chapter, we extend the applicability of RQV to distributed self-adaptive systems. To this end, we introduce DECIDE, an RQV-driven approach for DEcentralised Control In Distributed sElf-adaptive software. DECIDE addresses two of the key research objectives identified in a recent research roadmap for self-adaptive systems [56]:

- *Decentralisation of control loops.* This eliminates the single point of failure created by the use of a centralised control loop, improves the flexibility of self-adaptive systems, and is in line with the original autonomic computing vision [139].

- *Practical runtime verification and validation.* DECIDE supports the adoption of runtime verification in self-adaptive software, by drastically reducing control loop overheads through replacing the system-level RQV of very large models with the distributed, component-level RQV of models many orders of magnitude smaller.

To the best of our knowledge, DECIDE is the first approach that uses formal verification to simultaneously decentralise the control loop of self-adaptive systems, and to provide guarantees on their compliance with QoS requirements.

DECIDE is applicable to distributed self-adaptive systems whose components share a set of common system-level QoS requirements and cooperate to achieve these requirements. Each component of a DECIDE system executes a decentralised control workflow comprising the following stages. First, in a *local capability analysis* stage, local RQV takes place to calculate a set of possible contributions that the component can make towards the realisation of the system-level QoS requirements. This stage is executed infrequently, i.e., when a component joins the system or after major environment and internal changes (e.g., a significant workload increase or a failure of component parts). Next, the component shares a capability summary, i.e., a finite set of these possible contributions, with its peer components. After calculating a new such summary locally or receiving one from a peer, the component performs a *selection of a local contribution-level agreement (CLA)*. This CLA is one of the alternative contributions from the capability summary of the local component. The key advantage of our approach is in this CLA selection, which is carried out such that the system QoS requirements are provably met as long as each component achieves or betters its CLA. Most of the time, the *execution of a local control loop* is the only DECIDE stage executed by each component. Its purpose is to ensure compliance with the selected component CLA by performing RQV-based local adaptation. Infrequently, components are unable to achieve their CLAs due to *major changes*. These events trigger a new local capability analysis, the sharing of capability summary with peers, and the selection of new CLAs.

The main contribution of this chapter is the DECIDE framework for the decentralisation of control loops of distributed self-adaptive software. We also propose an RQV method for devising component QoS capability summaries in distributed self-adaptive systems and a method for the decentralised selection of component contribution-level agreements. We describe these contributions in Section 5.1. In Section 5.2 we present the DECIDE implementation within the open-source platform MOOS-IvP, which was used to develop a simulated distributed embedded system in the unmanned underwater vehicles domain. In Section 5.3 we present the findings from our evaluation. We discuss related work and summarise DECIDE in Sections 5.4 and 5.5, respectively.

5.1 DECIDE

DECIDE is applicable to distributed systems whose components exhibit stochastic behaviour, and involves the runtime quantitative verification of stochastic models that describe the behaviour of these components. Similarly to our work from Chapters 3 and 4, DECIDE uses parametric Markov models to define the uncertainty associated with the system itself and the environment in which the system operates. Before describing the theoretical foundation of DECIDE, we introduce the distributed self-adaptive system that will be used to illustrate the application of DECIDE and for its evaluation.

Distributed Multi-UUV Embedded System

Consider a distributed multi-UUV (unmanned underwater vehicle) embedded system that extends our single-UUV system from Section 2.2.1.1. The n-UUV system is deployed on a surveillance and data collection mission. The $n > 1$ UUVs travel within proximity of each other, and the i-th UUV is equipped with $n_i > 0$ on-board sensors that can take periodic measurements of a characteristic of the ocean environment (e.g., dissolved oxygen, salinity or temperature). The l-th sensor of UUV i operates with varying rate $r_{il} \geq 0$, and the probability p_{il} that one of its measurements is sufficiently accurate for the purpose of the mission depends on the UUV speed $sp_i \in [0, sp_i^{max}]$. This is typical for such devices, e.g., the measurement error of sonars can be approximated by a normal distribution with zero mean and standard deviation that increases with speed [1]. For each measurement taken, an amount of energy e_{il} is consumed. Finally, each UUV can switch on and off its sensors individually (e.g., to save battery power when not required), but each of these operations consumes energy given by e_{il}^{on} and e_{il}^{off}, respectively. To complete its mission successfully, the multi-UUV system must meet the system-level QoS requirements from Table 5.1.

In addition to these system-level QoS requirements, each UUV i must satisfy the local QoS requirements from Table 5.2. In a dynamic environment (like the ocean environment), each UUV i should adapt to changes in the operating rates of its sensors and to sensor failures, by continually adjusting:

- the UUV speed sp_i;
- the sensor configuration $x_{i1}, x_{i2}, ..., x_{in_i}$ (where $x_{il} = 1$ if the l-th sensor is on and $x_{il} = 0$ otherwise)

so that the system-level and UUV-level QoS requirements are satisfied at all times.

[1] for example, `http://www.ashtead-technology.com/rental-equipment/rdi-300khz-navigator`

Table 5.1: System-level QoS requirements for the multi-UUV distributed system

ID	Informal description
R1	"The n UUVs should take at least 1000 measurements of sufficient accuracy per 60 seconds of mission time."
R2	"At least two UUVs should have switched-on sensors at any time."
R3	"If requirements R1 and R2 are satisfied by multiple configurations, the system should use one of these configurations that minimises the energy consumption (so that the mission can continue for longer)"

Table 5.2: UUV-level QoS requirements for UUV i from the multi-UUV distributed system

ID	Informal description
R4	"The energy consumed by the UUV sensors must not exceed e_i^{max} Joules per 60 seconds of mission time."
R5	"The UUV should use only sensors whose measurements are accurate with probability at least p_i^{min}."
R6	"If requirements R4 and R5 are satisfied by multiple configurations, the UUV should use one of these configurations that maximises its speed (so that the mission can complete earlier) and minimises the local energy consumption (so that the mission can continue for longer), given by the cost function $w_1 e_i + w_2 sp_i^{-1}$, where w_1 and w_2 are UUV-specific weights."

5.1.1 Formal Description of a DECIDE System

DECIDE distributed self-adaptive systems comprise $n > 1$ components that cooperate to achieve a set of common objectives. Each component within DECIDE executes the self-adaptation workflow shown in Figure 5.1. We use Cfg_i and Env_i to denote the set of possible configurations and the set of possible environment states for the i-th component, respectively. Thus Cfg_i corresponds to parameters that the local control loop of component i can modify, and Env_i represent parameters that the component can only observe. Additionally, the i-th component has $m_i \geq 1$ QoS attributes $attr_{i1} \in V_1, attr_{i2} \in V_2, \ldots, attr_{im_i} \in V_{m_i}$, where the value domain V_j of the j-th attribute could be $\mathbb{R}, \mathbb{R}_+, \mathbb{B} = \{true, false\}$ etc. The m_i QoS attributes are classified into the following types (Figure 5.2): i) system-level QoS requirements; ii) system-level cost; iii) local-level (i.e., component-specific) QoS requirements; and iv) local-level cost, and satisfy the following conditions:

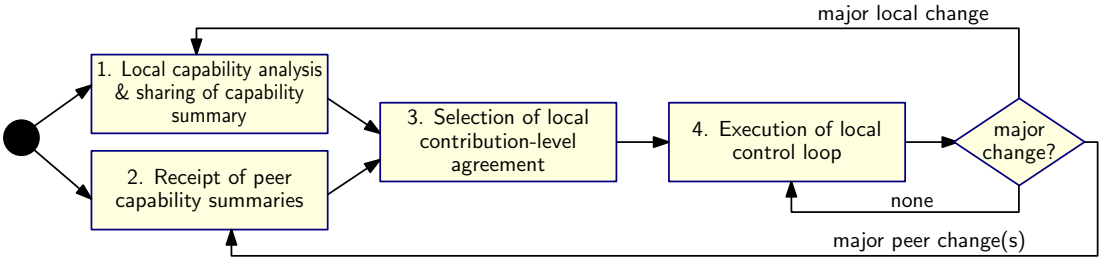


Figure 5.1: Decentralised self-adaptation workflow of a DECIDE component.

1. For any component i, the value of its j-th QoS attribute depends on the current configuration $c \in \mathit{Cfg}_i$ and the current environment state $e \in \mathit{Env}_i$, and can be obtained through the quantitative verification of the following

$$\mathit{attr}_{ij}(e, c) = f_i j(e, c, M_i(e, c) \models \Phi_{ij}) \tag{5.1}$$

   where M_i is a Markov model parametrised by the state of the environment the component operates in and the configuration selected by its local control loop, Φ_{ij} is a probabilistic temporal logic formula, and $f(\cdot, \cdot, \cdot)$ is a function that can be evaluated in $O(1)$ time.

2. Attributes $\mathit{attr}_{i1}, \mathit{attr}_{i2}, \ldots, \mathit{attr}_{im}, m < m_i$, are associated with the $m > 0$ system-level QoS requirements of the DECIDE distributed system. Formally, the j-th system QoS requirement, $1 \leq j \leq m$, is specified as

$$\mathit{expr}_j(\mathit{attr}_{1j}, \mathit{attr}_{2j}, \ldots, \mathit{attr}_{nj}) \bowtie_j \mathit{bound}_j \tag{5.2}$$

   where a non-exhaustive list of options for the expression expr_j, relational operator $\bowtie_j$ and bound bound_j is shown in Table 5.3.

3. Attribute $\mathit{attr}_{i,m+1}$ is a measure of the system-level cost associated with the current environment state and configuration of component i. Accordingly, $V_{m+1} = \mathbb{R}_+$ and the system-level cost $\sum_{i=1}^{n} = \mathit{attr}_{i,m+1}$ needs to be minimised subject to

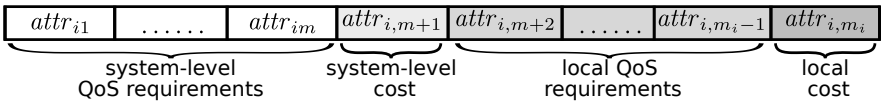


Figure 5.2: QoS attributes of a DECIDE component and their roles in defining system- and local-level QoS requirements.

Table 5.3: Categories of DECIDE system-level QoS requirements from (5.2)

V_j	$expr_j(attr_{1j}, attr_{2j}, \ldots, attr_{nj})$	$\bowtie_j \in$	$bound_j \in$	Types of QoS requirements
$\mathbb{R}_+$	$\sum_{i=1}^{n} w_i attr_{ij}$, $w_i > 0$ weights	$\{<, \leq, \geq, >\}$	$\mathbb{R}_+$	throughput, energy usage, response time
$[0,1]$	$\prod_{i=1}^{n} w_i attr_{ij}$, $w_i > 0$ weights	$\{<, \leq, \geq, >\}$	$[0,1]$	reliability, availability
$\mathbb{B}$	$booleanExpr(attr_{1j}, attr_{2j}, \ldots, attr_{nj})$	$\{=, \neq\}$	$\mathbb{B}$	liveness, security

the m QoS requirements being satisfied.

4. Attributes $attr_{i,m+2}, attr_{i,m+3}, \ldots, attr_{i,m_i-1}$ represent local-level QoS requirements. We have $V_{m+2} = V_{m+3} = \ldots = V_{m_i-1} = \mathbb{B}$, and the local requirements are satisfied iff

$$attr_{ij} = true \text{ for } j = m+2, m+3, \ldots, m_i - 1. \tag{5.3}$$

5. Attribute $attr_{i,m_i} \in \mathbb{R}_+$ represents local cost associated with the current environment state and configuration of component i. This value needs to be minimised, subject to system and local QoS requirements being satisfied.

Example 5.1. The set of configurations for UUV i of our distributed n-UUV system is $Cfg_i = Sp_i \times \{0,1\}^{n_i}$, where $(sp_i, x_{i1}, x_{i2}, \ldots, x_{in_i}) \in Cfg_i$ give the UUV speed sp_i and sensor configurations $x_{i1}, x_{i2}, \ldots, x_{in_i}$ selected by the local control loop. The set of environment states for UUV i is $Env_i = \mathbb{R}_+^{n_i}$, where $(r_{i1}, r_{i2}, \ldots, r_{in_i}) \in Env_i$ gives the measurement rates for the n_i sensors.

We use the CTMC model in Figure 2.3 to model the l-th sensor of the i-th UUV and denote this model M_{il}[2]. The Markov model $M_i(e, c)$ used to compute the QoS attributes of UUV i in (5.1) is obtained through the parallel composition of the n_i sensor models: $M_i = M_{i1} \parallel M_{i2} \parallel \ldots \parallel M_{in_i}$.

Given the model M_i, the CSL formulae and the functions in Table 5.4 are used in (5.1) to establish the QoS attributes for requirements R1–R6. The $m = 2$ system-level

[2] The indices of the model parameters from Figuree 2.3 are adjusted accordingly to suit the n-UUV system; thus, the configurable parameter x_i becomes x_{il}, the sensor rate r_i becomes r_{il}, etc.

Table 5.4: QoS attributes for UUV i, where val_{ij} is the value of $M_i(e,c) \models \Phi_{ij}$

j	V_j	Φ_{ij}	$attr_{ij} = f_{ij}(e, c, val_{ij})$
1	$\mathbb{R}_+$	$R^{\text{“}energy\text{”}}_{=?}\left[C^{\leq 60}\right]$	val_{i1}
2	$\mathbb{B}$	$P_{\geq 1}\left[F\ \mathsf{on}_{i1} \vert \mathsf{on}_{i2} \vert \ldots \vert \mathsf{on}_{in_i}\right]$	val_{i2}
3	$\mathbb{R}_+$	$R^{\text{“}energy\text{”}}_{=?}\left[C^{\leq 60}\right]$	val_{i3}
4	$\mathbb{B}$	$R^{\text{“}energy\text{”}}_{\leq e_i^{\max}}\left[C^{\leq 60}\right]$	val_{i4}
5	$\mathbb{B}$	$\bigwedge_{l=1}^{n_i}\left(\mathsf{read}_{il} \Rightarrow P_{\geq p_i^{\min}}\left[X\ \mathsf{accurate}_{il}\right]\right)$	val_{i5}
6	$\mathbb{B}$	$R^{\text{“}energy\text{”}}_{=?}\left[C^{\leq 60}\right]$	$w_1 val_{i6} + w_2 sp^{-1}$

requirements, R1 and R2, are given by the following instances of (5.2):

$$\begin{aligned} &\textbf{R1:} \sum_{i=1}^{n} attr_{i1} \geq 1000 \\ &\textbf{R2:} \bigvee_{1 \leq i_1 < i_2 \leq n} (attr_{i_1 2} \wedge attr_{i_2 2}) = true \end{aligned} \tag{5.4}$$

5.1.2 Stage 1: Local capability analysis

During this DECIDE stage, each component uses runtime quantitative verification to assemble a summary of its capabilities, as formally defined below.

Definition 5.1. *Given a DECIDE distributed system with the characteristics specified earlier, a finite set $CS_i \subset V_1 \times V_2 \times \cdots \times V_{m+1}$ is an* α-confidence capability summary *for the i-th system component iff for any $(a_{i1}, a_{i2}, \ldots, a_{i,m+1}) \in CS_i$ the local control loop of the component can ensure that:*

(i) $attr_{ij} \bowtie_j a_{ij}$, *for* $1 \leq j \leq m$

(ii) $attr_{i,m+1} \leq a_{i,m+1}$

(iii) $attr_{ij} = true$, *for* $m + 1 < j \leq m_i$

with probability at least $\alpha \in (0, 1)$.

The DECIDE method for calculating the α-confidence capability summary of the i-th system component, $1 \leq i \leq n$, involves the local execution of the steps below.

1. *Configuration analysis*— Select $N_i > 0$ disjoint configuration subsets Cfg_i^1, Cfg_i^2, $\ldots$, $Cfg_i^{N_i} \subset Cfg_i$ that correspond to different *modes of operation* for component i. What constitutes a mode of operation for a component is application dependent. Possible examples include running different numbers of component instances, or operating with different degrees of accuracy. As illustrated later in this section, for the UUV system from our running example, using different sets of sensors corresponds to different modes of operation for a UUV.

2. *Environment analysis*— Identify subsets of environment states Env_i^1, Env_i^2, $\ldots$, $Env_i^{N_i} \subseteq Env_i$ associated with the N_i configuration subsets, such that the probability that the actual environment state of the component is in Env_i^k is at least α, for any $1 \leq k \leq N_i$. These subsets can be identical. However, in the most general case, each configuration subset Cfg_i^k may render different areas of the environment state irrelevant, and DECIDE exploits this as illustrated in Example 5.2.

3. *Attribute analysis 1*— Check that for any $1 \leq k \leq N_i$ and for any $1 \leq j \leq m$ with $\bowtie_j \in \{=, \neq\}$, the QoS attribute $attr_{ij}(c, e)$ has a single value, a_{ij}^k, for all $(c, e) \in (Cfg_i^k, Env_i^k)$. When this is not the case, further partition the configuration set Cfg_i^k into disjoint subsets that satisfy this constraint. As shown in Table 5.3, one of the scenarios in which $\bowtie_j \in \{=, \neq\}$ is when $V_j = \mathbb{B}$. In this case, Cfg_i^k needs to be partitioned into two subsets. For other scenarios, (e.g., when $V_j = \mathbb{R}_+$), DECIDE can be applied only if this operation partitions Cfg_i^k into a finite (and usually small) number of subsets. The rationale for this operation is that we want to associate each configuration set Cfg_i^k with a "bound" a_{ij}^k for each $attr_{ij}$, $1 \leq j \leq m$, and the bounds a_{ij}^k are common values for QoS attributes $attr_{ij}$ with $\bowtie_j \in \{=, \neq\}$.

4. *Attribute analysis 2*— For all attributes $attr_{ij}$, $1 \leq j \leq m$, with $\bowtie_j \in \{<, \leq, \geq, >\}$, and for each configuration set Cfg_i^k, find simultaneous bounds $a_{ij}^k \in V_j$ such that

$$\forall e \in Env_i^k \bullet \exists c \in Cfg_i^k \bullet global(c, e) \wedge local(c, e), \tag{5.5}$$

where

$$global(c, e) = \bigwedge_{\substack{1 \leq j \leq m \\ \bowtie_j \notin \{=, \neq\}}} \left(attr_{ij}(c, e) \bowtie_j a_{ij}^k \right)$$

and

$$local(c,e) = \bigwedge_{m+2 \le j \le m_i - 1} attr_{ij}(c,e).$$

When there is a single system-level QoS attribute $attr_{ij}$ with $\bowtie_j \in \{<, \le, \ge, >\}$, its associated a_{ij}^k bound can be calculated as

$$a_{ij}^k = \begin{cases} \max\limits_{e \in Env_i^k} \min\limits_{\substack{c \in Cfg_i^k \\ local(c,e)}} attr_{ij}(c,e), & \text{if } \bowtie_j \in \{<, \le\} \\ \min\limits_{e \in Env_i^k} \max\limits_{\substack{c \in Cfg_i^k \\ local(c,e)}} attr_{ij}(c,e), & \text{otherwise} \end{cases} \tag{5.6}$$

Otherwise, a multi-objective optimisation technique such as [69, 81] needs to be used to calculate the a_{ij}^k values.

5. *Cost analysis*— Calculate the cost upper bound

$$a_{i,m+1}^k = \max_{e \in Env_i^k} \min_{\substack{c \in Cfg_i^k, \\ global(c,e) \wedge local(c,e)}} attr_{i,m+1}(c,e).$$

6. *Capability summary assembly*— Use the a_{ij}^k bounds from steps 3–5 to assemble

$$CS_i = \{cs_i^1, cs_i^2, \ldots, cs_i^{N_i}\}, \tag{5.7}$$

where $cs_i^k = (a_{i1}^k, a_{i2}^k, \ldots, a_{i,m+1}^k)$, $1 \le k \le N_i$.

Theorem 1. *The set CS_i in (5.7) is an α-confidence capability summary for component i of a DECIDE system.*

Proof. We show that for any $cs_i^k = (a_{i1}^k, a_{i2}^k, \ldots, a_{i,m+1}^k) \in CS_i$, the local control loop of component i can adjust the configuration of the component such that properties (i)–(iii) from Definition 5.1 are satisfied with probability at least α. For $1 \le j \le m$, the selection of a_{ij}^k in *Attribute analysis 1–2* ensures that, for any environment state $e \in Env_i^k$:

- $attr_{ij} \bowtie_j a_{ij}$ for all configurations in Cfg^k if $\bowtie_j \in \{=, \ne\}$;
- there is an environment-dependent configuration $c \in Cfg^k$ (given by (5.5)), such that $attr_{ij} \bowtie_j a_{ij}$ simultaneously for the other $attr_{ij}$ attributes.

By always selecting this configuration c, the local control loop can ensure that property (i) is satisfied whenever $e \in Env_i^k$, which happens with probability at least α (cf. the *Environment analysis* step). The *Cost analysis* step ensures that $a_{i,m+1}^k \geq attr_{i,m+1}(c,e)$ for all $e \in Env_i^k$, so the selection of configuration c also makes property (ii) satisfied with probability at least α. Finally, c satisfies $local(c,e)$, so using the configuration c ensures that property (iii) is satisfied with probability at least α too. □

At the end of the local capability analysis stage of DECIDE, the local capability summary (5.7) is shared with the other components within the distributed system. On distributed systems with reliable and high-bandwidth communication mechanisms, capability summary sharing is achieved using these mechanisms directly. For distributed systems with limited and/or unreliable inter-component communication capabilities, DECIDE uses recently emerged platforms for the engineering of distributed systems such as Kevoree [84] and DEECo [28]. This is the case for the UUV system from our running example.

Example 5.2. Suppose that the i-th UUV from our running example has $n_i = 2$ on-board sensors whose operating rates r_{i1} and r_{i2} are normally distributed with mean 2s^{-1} and standard deviation 0.2s^{-1}, and with mean 4s^{-1} and standard deviation 0.3s^{-1}, respectively. The UUV_i environment state has the form (r_{i1}, r_{i2}), and the set of all environment states is $Env_i = [0,\infty]^2$. Also, assume that the UUV speed sp_i can be adjusted in the range $[1\text{m/s}, 5\text{m/s}]$. Hence, the UUV configuration set is $Cfg_i = [1,5] \times \{0,1\}^2$, where for any configuration $(sp_i, x_{i1}, x_{i2}) \in Cfg_i$, $x_{ij} = 1$ if sensor j is switched on and $x_{ij} = 0$ otherwise, for $j \in \{1,2\}$. Finally, suppose that the bounds for local QoS requirements R4 and R5 are $e_i^{\max} = 1000J$ and $p_i^{\min} = 0.9$, and that the energy used by the sensor operations are: $e_{i1} = 3J$, $e_{i1}^{\text{on}} = 15J$, $e_{i1}^{\text{off}} = 3J$, $e_{i2} = 2J$, $e_{i2}^{\text{on}} = 10J$, $e_{i2}^{\text{off}} = 2J$. The DECIDE instance running on UUV_i assembles an ($\alpha = 0.95$)-confidence capability summary as follows:

1. *Configuration analysis*— A UUV mode of operation corresponds to using different subsets of sensors, so there are four configuration subsets: $Cfg_i^1 = \{(sp_i, 0, 0) | sp_i \in [1,5]\}$, $Cfg_i^2 = \{(sp_i, 1, 0) | sp_i \in [1,5]\}$, $Cfg_i^3 = \{(sp_i, 0, 1) | sp_i \in [1,5]\}$ and $Cfg_i^4 = \{(sp_i, 1, 1) | sp_i \in [1,5]\}$.

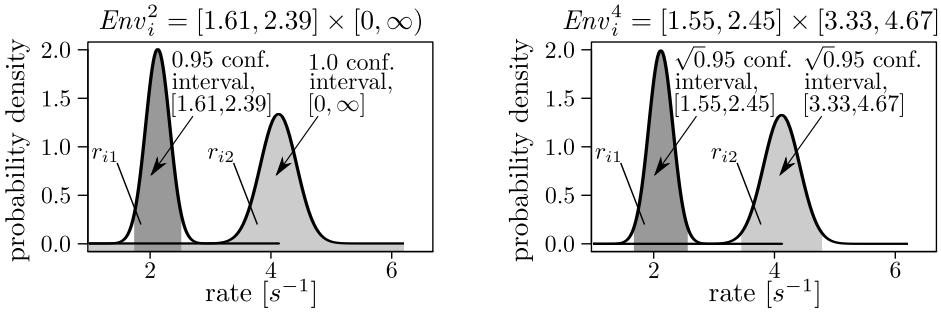


Figure 5.3: Environment analysis $Env_i^2 = [1.61, 2.39] \times [0, \infty]$ and $Env_i^4 = [1.55, 2.45] \times [3.33, 4.67]$ for configuration subsets Cfg_i^2 and Cfg_i^4 for a two-sensor UUV.

2. *Environment analysis*— Assuming that the sensor rates r_{i1} and r_{i2} are independent of each other, the environment state subsets Env_i^k, $1 \leq k \leq 4$, are obtained as the Carthesian product of α_1 and α_2 confidence intervals for r_{i1} and r_{i2}, respectively, where $\alpha_1\alpha_2 = \alpha = 0.95$. If a sensor is switched off for a configuration subset Cfg_i^k, the confidence level associated with this sensor (α_1 or α_2) is set to 1.0 when calculating Env_i^k. This allows the use of a smaller confidence level for the other sensor, which is potentially active. The result is a narrower confidence interval for the rate of active sensors, and therefore a capability summary that reflects better the actual ability of the UUV. Informally, the UUV can "promise" a stronger contribution to achieving the system requirements for a configuration subset Cfg_i^k if it disregards the state of the sensors switched off for the configurations in Cfg_i^k. Figure 5.3 summarises the calculation of $Env_i^2 = [1.61, 2.39] \times [0, \infty]$ and $Env_i^4 = [1.55, 2.45] \times [3.33, 4.67]$ for configuration subsets Cfg_i^2 and Cfg_i^4, respectively.

3. *Attribute analysis 1*— The relational operators for the $m = 2$ system-level QoS requirements (5.4) are $\bowtie_1$='$\geq$' and $\bowtie_2$='=', so DECIDE checks that the second attribute from Table 5.4 takes a single value within each configuration subset Cfg_i^k, $1 \leq k \leq 4$. This check is successful because $attr_{i2} = false = a_{i2}^1$ for all configurations in Cfg_i^1 (since both sensors are switched off) and $attr_{i2} = true = a_{i2}^k$ for all configurations in Cfg_i^k, $2 \leq k \leq 4$. Hence, no further partition of any configuration subset Cfg_i^k is required.

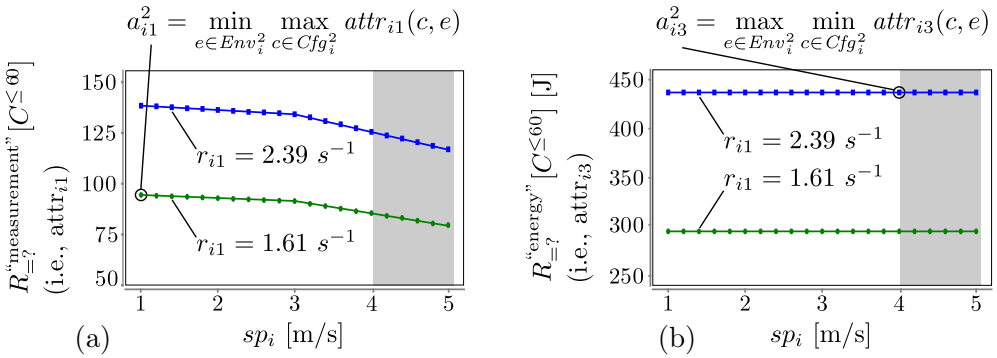


Figure 5.4: Verification of Φ_{i1} and Φ_{i3} from Table 5.4; shaded areas correspond to configurations that violate local requirement R5.

4. *Attribute analysis 2*— Requirement R1 in (5.4) is the only system-level requirement whose associated relational operator $\bowtie_1$ belongs to the set $\{<, \leq, \geq, >\}$. Accordingly, DECIDE uses runtime quantitative verification to derive the bounds a^k_{i1} in (5.6) for $1 \leq k \leq 4$. Figure 5.4(a) illustrates the analysis carried out to establish a^2_{i1} using the probabilistic model checker PRISM [149]. The minimum number of (accurate) measurements $attr_{i1}$ is obtained for the lowest measurement rate in Env^2_i, i.e., $r_{i1} = 1.61\text{s}^{-1}$; the bound a^2_{i1} corresponds to this rate, and to the most advantageous configuration, i.e., $sp_i = 1\text{m/s}$.

5. *Cost analysis*— As shown by the runtime quantitative verification results in Figure 5.4(b), the cost $attr^k_{i3}$ is constant for each environment state in Env^k_i. Hence, the maximum cost associated with the k-th configuration subset, a^k_{i3}, corresponds to the highest sensor rate in Env^k_i, i.e., $r_{i1} = 2.39s^{-1}$.

6. *Capability summary assembly*— The bounds a^k_{ij}, $1 \leq j \leq 3$, $1 \leq k \leq 4$, obtained in steps 3–5 are organised into the four-element capability summary $CS_i = \{(0, \mathit{false}, 5), (93, \mathit{true}, 433), (192, \mathit{true}, 532), (278, \mathit{true}, 984)\}$. Each summary element corresponds to a set of values for the system-level QoS requirements from Table 5.1. For instance, the element $cs^2_i = (93, \mathit{true}, 433)$ specifies that the i-th UUV using configuration Cfg^2_i can do 93 accurate measurements (R1), at least one of its sensors is switched on (R2) and for this operation it consumes $433J$ (R3). These values have been extracted after executing steps 3–5 (see Figure 5.4). The derivation of the other capability summary elements follows similar reasoning.

5.1.3 Stage 2: Receipt of Peer Capability Summaries

In this DECIDE stage, the α-confidence capability summary (5.7) of a component is shared with all other components within the distributed system. Major changes, which are presented formally in Section 5.1.6, occur with a frequency that depends on the confidence level α. For large α values, capability summaries are highly conservative and will rarely need updating. Local control loops can achieve any element of such capability summaries except after severe failures or significant environment changes, and the frequency with which capability summaries need updating is orders of magnitude lower than the frequency with which decisions are made by the local control loops. As a downside, components operate conservatively. Local control loops may overachieve the agreed CLAs most of the time, and component-level costs may be higher than necessary. In contrast, the selection of a lower α allows components to operate cost effectively and closer to their top ability. However, capability summaries are invalidated by smaller changes, and need to be recalculated with higher frequency. A suitable confidence level α is one that provides a good trade-off between the two scenarios. We show later in evaluation (Section 5.3) that what constitutes a good trade-off is system dependent.

Major changes experienced by different components may not be independent of each other. As an example, the same environment change may affect several components, leading to multiple major changes within a short time of each other. DECIDE handles this scenario by using a small time window to group together related capability summary updates. These time windows are started by the receipt of an update, and a CLA selection stage for all updates received within the window is triggered when the window ends. To ensure that updates due to the same major change are handled together with high likelihood, the width t_{w} of the window is set to

$$t_{\mathrm{w}} \approx \max(t_{\mathrm{detect}} + t_{\mathrm{CS}} + t_{\mathrm{comm}}) - \min(t_{\mathrm{detect}} + t_{\mathrm{CS}}), \tag{5.8}$$

where t_{detect}, t_{CS}, t_{comm} are estimates of the times required for a component to detect a major change, to recalculate the capability summary, and to communicate it to peer components, respectively. The second term of (5.8) does not include t_{comm} because components affected by a major change "receive" their own capability summaries without a communication step.

DECIDE does not suggest a new mechanism for sharing the α-confidence peer capability summaries. This DECIDE stage exploits the data sharing capabilities of recently emerged platforms for the engineering of distributed systems such as Kevoree [84] and DEECo [28].

5.1.4 Stage 3: Selection of Component Contributions

In this stage of the DECIDE self-adaptation workflow, the system components decide their contributions to the realisation of the system-level QoS requirements. To this end, they use their capability summaries CS_1, CS_2, ..., CS_n to solve the optimisation problem:

$$\begin{array}{ll} \text{minimise} & \sum_{i=1}^{n} a_{i,m+1} \\ \text{subject to} & expr_j(a_{1j}, a_{2j}, \ldots, a_{nj}) \bowtie_j bound_j, 1 \leq j \leq m \\ \text{and} & (a_{i1}, a_{i2}, \ldots, a_{i,m+1}) \in CS_i, 1 \leq i \leq n \end{array} \tag{5.9}$$

Assuming the problem has a solution, the CLA for the i-th component is given by

$$cla_i = (a_{i1}, a_{i2}, \ldots, a_{i,m+1}) \tag{5.10}$$

from this solution, and we say that the i-th system component satisfies its CLA iff the QoS attributes of the component satisfy for all $1 \leq j \leq m$

$$\begin{array}{ll} attr_{ij} \bowtie_j a_{ij}, & \text{if } \bowtie_j \in \{<, \leq, \geq, >\} \\ attr_{ij} = a_{ij}, & \text{otherwise (i.e., if } \bowtie_j \in \{=, \neq\}) \end{array} , \tag{5.11}$$

Remember that the local capability analysis stage of DECIDE ensures that component configurations that satisfy (5.11) exist with probability at least α.

Theorem 2. *Let cla_1, cla_2, ..., cla_n be the CLAs (5.10) of a DECIDE system with QoS requirements (5.2). If component i satisfies cla_i for all $1 \leq i \leq n$, then the system QoS requirements are satisfied.*

Proof. Suppose that the n components of a DECIDE system satisfy the CLAs (5.10), and consider the j-th system-level QoS requirement (5.2), $1 \leq j \leq m$. Then $attr_{ij} \bowtie_j a_{ij}$ for all components i, $1 \leq i \leq n$. We will prove that the j-th system-level QoS requirement is satisfied by examining each entry in Table 5.3 individually.

For the first entry, $\bowtie_j \in \{<, \leq, \geq, >\}$, so $expr_j$ satisfies

$$expr_j(attr_{1j}, \ldots, attr_{nj}) = \sum_{i=1}^{n} w_i attr_{ij} \bowtie_j \sum_{i=1}^{n} w_i a_{ij} =$$

$$= expr_j(a_{1j}, \ldots, a_{nj}) \bowtie_j bound_j,$$

and the j-th QoS requirement is satisfied.
For the second entry, $attr_{ij} \in [0, 1]$, so the same reasoning can be applied to show that the requirement is satisfied:

$$expr_j(attr_{1j}, \ldots, attr_{nj}) = \prod_{i=1}^{n} w_i attr_{ij} \bowtie_j \prod_{i=1}^{n} w_i a_{ij} =$$

$$= expr_j(a_{1j}, \ldots, a_{nj}) \bowtie_j bound_j.$$

Finally, for the third entry in Table 1, $\bowtie_j \in \{=, \neq\}$, so $attr_{ij} = a_{ij}$ for all $1 \leq i \leq n$, thus

$$expr_j(attr_{1j}, \ldots, attr_{nj}) = expr_j(a_{1j}, \ldots, a_{nj}) \bowtie_j bound_j,$$

which completes the proof. □

DECIDE does not prescribe how the optimisation problem (5.9) should be solved, as this is application specific. Depending on the nature of the DECIDE system and its requirements, the best way to obtain the component CLAs (5.10) may be by using an efficient dynamic programming or greedy algorithm, a metaheuristic or, when the solution space $CS_1 \times CS_2 \times \ldots \times CS_n$ is sufficiently small, using brute-force. The CLA calculation is performed independently by each system component (using the same deterministic method). Although DECIDE is sufficiently generic to allow the execution of this calculation by a "leader" component that would then communicate it to all other components, the independent calculation is preferred because it avoids a single point of failure and additional communication between components. This is done, of course, at the expense of duplicating the CLA selection on all components.

Example 5.3. Suppose that the distributed UUV system from our running example comprises 3 UUVs, i.e., $n = 3$, each equipped with 2 on-board sensors and the characteristics shown in Table 5.5. The capability summaries CS after completing the DECIDE stages described in Sections 5.1.2 and 5.1.3 are shown in Table 5.6. The instance of the optimisation problem (5.9) solved by the DECIDE module running on each UUV is

$$\begin{aligned} &\text{minimise} && \textstyle\sum_{i=1}^{n} a_{i3} \\ &\text{subject to} && \textstyle\sum_{i=1}^{n} a_{i1} \geq 1000 \\ & && \textstyle\bigvee_{1 \leq i_1 < i_2 \leq n} (a_{i_1 2} \wedge a_{i_2 2}) = \mathit{true} \\ &\text{and} && (a_{i1}, a_{i2}, a_{i3}) \in CS_i, 1 \leq i \leq 3 \end{aligned} \tag{5.12}$$

The optimisation problem for this system corresponds to a multiple-choice knapsack problem (MCKP) [137]. In this particular problem, however, we deal with the minimisation form of the MCKP, where the optimal solution is given by the minimum value of the objective function $\sum_{i=1}^{n} a_{i3}$, subject to a set of constraints being satisfied, of which at least one is upper bounded.

Our implementation of DECIDE described in Section 5.3 solves this problem by first transforming the minimisation form of MCKP to its equivalent maximisation problem by calculating for each UUV, the following values:

$$\begin{aligned} \bar{a_{i1}} &= max_{1 \leq k \leq N_i} a_{i1}^k \\ \bar{a_{i3}} &= max_{1 \leq k \leq N_i} a_{i3}^k \end{aligned} \tag{5.13}$$

and updating each capability summary cs_i^k, $1 \leq i \leq 3$, $1 \leq k \leq 4$, as follows:

$$\begin{aligned} a'_{i1} &= \bar{a_{i1}} - a_{i1}^k \\ a'_{i3} &= \bar{a_{i3}} - a_{i3}^k \\ C &= \textstyle\sum_{i=1}^{n} \bar{a_{i1}} - bound_1 \end{aligned} \tag{5.14}$$

Table 5.5: Characteristics of the three-UUV system

UUV i	$r_{i1}[Hz]$	$e_{i1}[J]$	$e_{i1}^{\text{on}}[J]$	$e_{i1}^{\text{off}}[J]$	$r_{i2}[Hz]$	$e_{i2}[J]$	$e_{i2}^{\text{on}}[J]$	$e_{i2}^{\text{off}}[J]$
1	4	1.3	10	2	4.5	1	5	1
2	3.5	1.6	15	3	4	1.3	10	2
3	4.5	1.3	10	2	5	1	5	1

where $bound_1$ is the constraint specified for system-level QoS requirement R1 and C is the knapsack capacity. Then using an efficient $O(n^2)$ dynamic programming algorithm, adapted from [157], we obtain the optimal solution to the problem solving the following equation recursively

$$DP[i][l] = max(l - a_{i1}^k \geq 0?DP[i-1][l - a_{i1}^k] + a_{i3}^k : -\infty) \tag{5.15}$$

where DP is a suitable data structure of size $4 \times C$, $1 \leq l \leq C$, and "?:" is the ternary operator (shortcut for "if...then...else"). Note that $DP[0][l], 1 \leq l \leq C$ is an auxiliary element of dynamic programming paradigm where all entries are 0.

After solving (5.15), the chosen CLAs (i.e., the optimal solution) for the 3-UUVs are: cla_1=(384, $true$, 647); cla_2=(190, $true$, 270); and cla_3 = (441, $true$, 707). These CLAs have been established using configuration subsets Cfg_1^4, Cfg_2^3, and Cfg_3^4, respectively.

Table 5.6: Capability summaries of the three-UUV system

k	cs_1^k	cs_2^k	cs_3^k
1	(0, false, 3)	(0, false, 5)	(0, false, 3)
2	(185, true, 350)	(163, true, 392)	(213, true, 381)
3	(208, true, 292)	(190, true, 270)	(236, true, 321)
4	(384, true, 467)	(343, true, 667)	(441, true, 707)

5.1.5 Stage 4: Execution of Local Control Loop

Most of the time, this is the only stage of the DECIDE self-adaptation workflow executed by the system components. The local control loop of a component ensures that the component complies with its CLA and local QoS requirements. To this end, DECIDE local control loops implement the established approach to achieving self-adaptation in (single control loop) software systems by using RQV. We presented RQV in Section 2.2.1 and used it successfully in our research introduced in Chapters 3 and 4.

For the local control loop of component i, the approach involves the runtime execution of the verification step (5.1) to establish the value of $M_i(e,c) \models \Phi_{i,j}, i \leq j \leq m_i$, either periodically and/or after events associated with environment or component changes. The aim is to verify if the QoS attributes of the component continue to satisfy the component CLA (5.10) and local requirements (5.3), and, if this is not the case, to identify a new configuration that does. DECIDE local control loops start the search for such new configurations with the configuration subset Cfg_i^k associated with the current component CLA, cla_i (recall that cla_i is an element from the capability summary CS_i of component i, and that each CS_i element corresponds to a configuration subset $Cfg_i^k \subset Cfg_i$). When no configuration in Cfg_i^k is suitable, the search is extended to the entire configuration space Cfg_i. This could work due to the conservative nature of capability summaries—configurations in $Cfg_i \setminus Cfg_i^k$ may better their α-confidence capabilities for certain environment states. Finally, if no configuration is available that satisfies the component CLA and local requirements, we say that component i is affected by a "major change", and its capability summary is recalculated (Section 5.1.2).

Example 5.4. Consider our distributed UUV system, and its two-sensor UUV_i we analysed in Example 5.2. Suppose that the CLA selected for UUV_2 in the selection of component contributions stage of DECIDE was $(190, \mathit{true}, 270)$ from the capability summary established in Example 5.3. Accordingly, its local control loop will adjust the local configuration in response to changes in the sensor rates r_{21} and r_{22} such that the UUV achieves at least 190 accurate measurements and consumes at most 270J for each 60s of operation. Figure 5.5 depicts the results of the RQV carried out for this purpose if $r_{22} = 3.68\text{s}^{-1}$ using only the configurations from the set $Cfg_2^3 = \{(sp_2, 0, 1) | sp_2 \in [1, 5]\}$ that is associated with the current CLA; the value of r_{21} is irrelevant since sensor 1 is switched off. The energy usage from the local requirement R4 (obtained by verifying Φ_{i3} from Table 5.4), is 224J for all configurations from Cfg_2^3. Hence, the CLA is satisfied

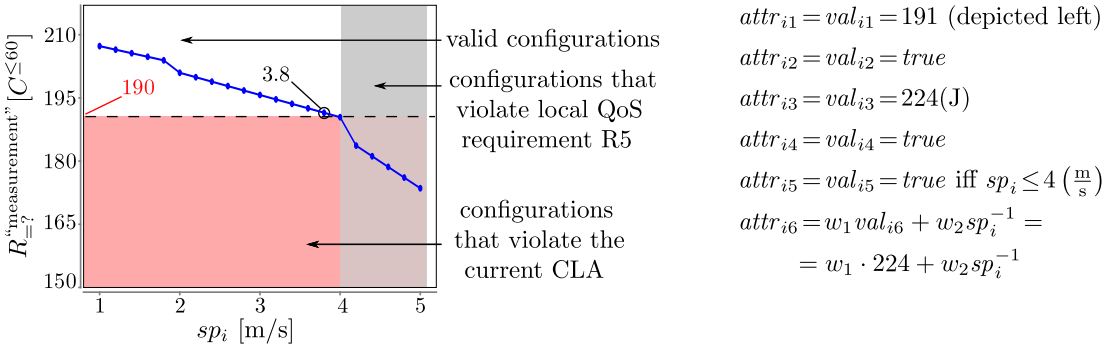


Figure 5.5: RQV of Φ_{i1} – Φ_{i6} from Table 5.4.

for multiple configurations in Cfg_2^3, and no configuration in $Cfg_2 \setminus Cfg_2^3$ is examined for this scenario. Given these results, the local control loop selects the configuration $(sp_2, x_1, x_2) = (3.8, 0, 1)$, which meets the CLA and local requirements, and minimises the local cost attr_{26}.

5.1.6 Stage 5: Major Changes

Major changes are environment or component changes that require the execution of other stages of the DECIDE self-adaptation workflow than the local control loop. DECIDE components deal with local and peer major changes.

A local major change occurs within the i-th component when:

(a) The local control loop cannot find a configuration that satisfies the component CLA and local requirements for the current environment state. This can be due to the environment state being outside the α-confidence area identified in the local capability analysis, or due to unexpected environment changes that invalidate this analysis.

(b) Internal changes such as failures of parts of the component make certain modes of operation (i.e., configuration subsets Cfg_i^k) unavailable.

(c) The capability summary CS_i is overly conservative. This may be due to an environment that is more favourable than the one considered in the local capability analysis, or to an extension to the configuration space of the component (e.g., when a previously failed part becomes operational).

In these scenarios, component i returns to the local capability analysis stage of the

DECIDE self-adaptation workflow. This stage is re-executed with updated environment and/or configuration sets.

A peer major change occurs when another component: (a) joins the system; (b) undergoes a local major change; or (c) leaves the system. Component i learns about peer major changes of types (a) and (b) when it receives a new capability summary. For the last type of peer major change, DECIDE requires that component failures are identified and announced by the communication and synchronisation platform underpinning the interactions between system components. This capability is already supported by platforms such as Kevoree [84] and DEECo [28], and can be readily exploited by DECIDE.

5.2 Implementation

To evaluate DECIDE, we developed a fully-fledged simulator for the multi-UUV self-adaptive system we introduced in Section 5.1. Like our previous work using a single UUV (Section 3.2), we used the open-source MOOS-IvP middleware [20].

For the multi-UUV system, we developed a DECIDE MOOS application (Figure 5.6) that implements the DECIDE stages depicted in Figure 5.1 and described in Sections 5.1.2 – 5.1.6. Note that the DECIDE local control loop from our new MOOS application reuses a small amount of code from our existing implementation of single-UUV self-adaptive system whose centralised control loop is also driven by RQV (Section 3.2). Nevertheless, the code for other DECIDE stages—local capability analysis, receipt of peer capability summaries, selection of component contributions, and major change identification within the local control loop—is entirely new. This new code represents over 90% of our DECIDE MOOS application.

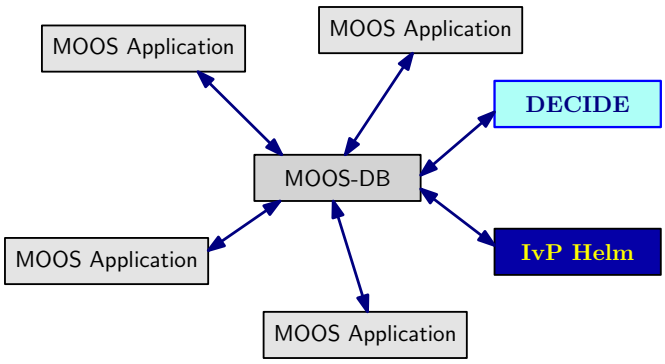


Figure 5.6: MOOS architecture, adapted from [20], including our DECIDE component.

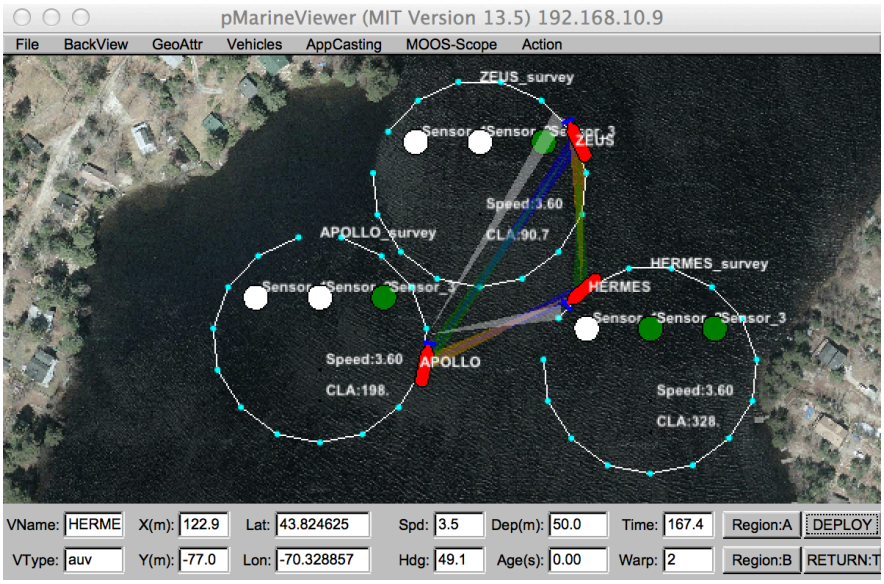


Figure 5.7: Self-adaptive multi-UUV system simulator.

Figure 5.7 shows a screenshot of a three-UUV instance of our self-adaptive UUV simulator. The screenshot depicts the time moment when the UUV code-named APOLLO notifies its peer UUVs ZEUS and HERMES that it incurred a major change by sending its new capability summary, and thus initiating a new execution of the DECIDE stage involving the selection of component contributions. The open-source code for our DECIDE MOOS application and multi-UUV simulator, the full experimental results summarised in the following section, additional information about DECIDE and a video recording of the demo from which we extracted the screenshot in Figure 5.7 are freely available at `http://www-users.cs.york.ac.uk/~simos/DECIDE`.

5.3 Evaluation

5.3.1 Research Questions

The aim of our experimental evaluation was to answer the following research questions:

RQ1 (Validation): Can DECIDE enhance distributed systems with self-adaptive capabilities? This is the first research work that decentralises the RQV control loop of distributed self-adaptive systems. Therefore, we want to establish whether DECIDE can support dependable self-adaptation in multi-UUV systems.

RQ2 (Efficiency): How efficiently can DECIDE reconfigure a distributed self-adaptive software system? With this research question we want to study the (communication and CPU) overheads incurred by DECIDE when a reconfiguration of the multi-UUV system is required. We also analyse any "loss" in efficiency by comparing the CLAs selected by DECIDE (cf. DECIDE stage 3) to those chosen by an "ideal" system that has perfect knowledge.

RQ3 (Confidence level α): How does the α-confidence level affect the adaptation behaviour of DECIDE? We used this research question to study the impact of the α-confidence level in assembling the capability summaries of system components. We also investigated any connection points between the α values and the ability of DECIDE to handle changes locally.

RQ4 (Scalability): How well can DECIDE scale with distributed systems of different sizes? We used systems comprising up to 32 UUVs to assess whether DECIDE can cope with systems of different sizes. To this end, we examined the overheads associated with the various DECIDE stages for these system sizes.

5.3.2 Experimental Setup

To evaluate DECIDE, we carried out a broad range of experiments using the multi-UUV system implementation and the simulator described in Section 5.2.

Table 5.7 shows the characteristics of the analysed multi-UUV system variants used for the evaluation. The system characteristics varied in these experiments include the number of UUVs n and the number of UUV sensors n_i, $1 \leq i \leq n$, as well as the confidence level α used to assemble the UUV capability summaries in the local capability analysis stage (Section 5.1.2). To simulate realistic runtime behaviour of the system and to examine the impact of different types of failure, the experiments were seeded with failure patterns that consisted of failures of sensors, sudden significant reductions in sensor measurement rates (i.e., not following the distribution stated in the sensor specification) and failures of entire UUVs.

All the experiments were carried out using a standard 2.6GhZ Intel Core i5 Macbook Pro computer with 16GB of memory and running Mac OSX 10.9 64-bit.

Table 5.7: Characteristics of analysed multi-UUV system variants

Type	Details	
#UUVs	$n \in \{1, 2, 3, \ldots, 32\}$	
UUV speed	$sp_i \in [1m/s, 5m/s]$	$1 \leq i \leq n$
Sensors per UUV	$n_i \in \{1, 2, 3\}$	$1 \leq i \leq n$
Sensor rate	$r_{il} \in [1Hz, 10Hz]$	$1 \leq i \leq n, 1 \leq l \leq n_i$
α-confidence level	$\alpha \in \{0.90, 0.95, 0.99\}$	

5.3.3 Results and Discussion

RQ1 (Validation). We start the presentation of our experimental results with the analysis of a typical mission carried out by the three-UUV system in Figure 5.7, each UUV being equipped with three sensors. The UUV characteristics for this scenario are shown in Table 5.8.

The three-UUV system should satisfy the system-level requirements R1–R3, while each UUV should also satisfy the UUV-level requirements R4–R6, described in Tables 5.1 and 5.2, respectively. For the purpose of this particular mission, we set the bound for system-level requirement R1 (i.e., the minimum number of sufficiently accurate measurements every 60 seconds) to 1000, and specify the thresholds for UUV-specific requirements R4 (i.e., the maximum energy consumed by the sensors on each UUV every 60 seconds) and R5 (the minimum accuracy probability of each sensor) to $e_i = 1000$ Joules and $p_i^{\min} = 0.9$, respectively.

Figure 5.8 depicts the execution of the DECIDE stages by these UUVs over a 5000-second simulated time period. The circled numbers ①, ②, ③, ④ correspond to the DECIDE stages 1–4 described in Sections 5.1.2–5.1.5.

Some of the main operations executed by the three UUVs at different time moments t and the events triggering them are summarised below:

- $t \approx 0$s – The local capability analysis, receipt of peer summaries and CLA selection stages of the DECIDE workflow are carried out by each UUV. The initial CLAs established as $(384, \mathit{true}, 647)$, $(190, \mathit{true}, 270)$ and $(441, \mathit{true}, 707)$ are selected by Apollo, Zeus and Hermes, respectively.

- $t \approx 300$s – Apollo experiences a significant, but not critical, performance degradation of the currently active sensor 3. To continue satisfying its CLA, Apollo switches off sensor 2 and starts using sensor 1 in conjunction with sensor 3. Al-

Table 5.8: Characteristics of a specific scenario of a three-UUV system

UUV	**Characteristics**			
	sensor id	**rate** $[s^{-1}]$	**failure time interval [s] (start:finish)**	**degradation [%]** *
Apollo	r_{11}	5.0		
	r_{12}	4.0		
	r_{13}	4.5	300:400, 1000:1100, 4000:4200	55, 20, 55
Hermes	r_{21}	3.5		
	r_{22}	4.5	2000:2100, 4000:4200	87, 87
	r_{23}	4.0		
Zeus	r_{31}	5.0		
	r_{32}	4.5	3000:3100	20
	r_{33}	5.0	2000:2100, 4000:4200	80, 65

* degraded rate as a percentage of the nominal rate

though the new configuration consumes more energy, it still satisfies Apollo's CLA, and no notification is sent to peers.

- $t \approx 400$s – Apollo is affected by a local change where sensor 3 recovers and the UUV selects sensor 2 instead of sensor 1. Since Apollo's contribution complies with its CLA, the UUV continues its operation without any further changes.

- $t \approx 1000$s – Apollo undergoes a major local change where the measurement rate of sensor 3 decreases significantly. As it is unable to meet its CLA, it carries out a new local capability analysis, and sends to its peers an updated capability summary. The new CLAs established for the three UUVs are $(189, \mathit{true}, 355)$, $(396, \mathit{true}, 664)$, $(447, \mathit{true}, 715)$.

- $t \approx 1110$s – As sensor 3 on Apollo has recovered, the UUV notifies its peers that it can resume a stronger contribution by sending them an updated capability summary, and thus initiates a new CLA re-negotiation which results in the CLAs: $(388, \mathit{true}, 651)$, $(190, \mathit{true}, 270)$, $(447, \mathit{true}, 715)$.

- $t \approx 2000$s – Hermes and Zeus, experience a slight decrease in the measurement rates of sensor 2 and sensor 3, respectively. To continue complying with their

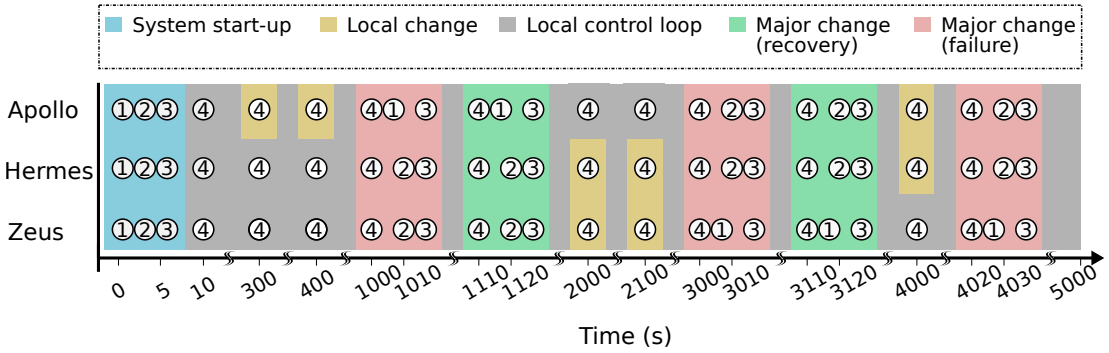


Figure 5.8: Execution of DECIDE stages 1–4 for a particular scenario including major changes and local sensor changes.

CLAs, both UUVs reduce their speed significantly (from 4 m/s and 3.9 m/s to 2.5 m/s and 3.5 m/s, respectively), while maintaining the current sensor configuration.

- $t \approx 2100$s – The sensors on Hermes and Zeus have recovered and the two UUVs increase their speed accordingly.

- $t \approx 3000$s – Sensor 2 on Zeus suffers from a serious degradation of service. The UUV is unable to fulfil its CLA, and as a result it calculates and shares with peers its new capability summary (with lower potential contributions). A new CLA selection takes place which results in the CLA combination $(388, \mathit{true}, 651)$, $(396, \mathit{true}, 664)$, $(240, \mathit{true}, 324)$.

- $t \approx 3110$s – The previously malfunctioning sensor 2 on Hermes becomes operational, and the UUV determines that its current capability summary is overly conservative. After sharing with peers the updated capability summary, the UUVs select the new CLAs $(388, \mathit{true}, 651)$, $(190, \mathit{true}, 270)$, $(448, \mathit{true}, 715)$, according to which Hermes contributes more in satisfying the system-level QoS requirements.

- $t \approx 4000$s – Apollo and Hermes are impacted by local changes and the measurement rate of sensor 3 on both UUVs decreases slightly. To continue satisfying their CLAs, Apollo switches sensor 2 off and starts using sensors 1 and 3 together, while Hermes decreases its speed significantly, from 4 m/s to 2.7 m/s.

- $t \approx 4020$s – Zeus experiences a major local change due to performance degradation on sensor 3. The UUV cannot find a configuration to satisfy its CLA and notifies its peers, sending its new capability summary to reconcile their CLAs, which results in the solution $(388, \mathit{true}, 651)$, $(396, \mathit{true}, 664)$, $(216, \mathit{true}, 386)$.

These results provide evidence that the multi-UUV system using DECIDE is able to successfully reconfigure itself in the presence of sensor failures, decrease in sensor measurement rates and total UUV failures. More specifically, if a DECIDE-enhanced UUV can handle locally sensor failures and find a configuration that meets its CLA (e.g., $t \approx 2000$s), no notification is sent to its peers. Thus, no other UUV is involved in the adaptation of the multi-UUV system. If, however, the failures are more significant (e.g., $t \approx 1000$s), then its peers are notified and a new CLA selection takes place to restore compliance with system-level QoS requirements. These observations indicate that DECIDE can deal effectively with both types of failures affecting a distributed system and, thus to support dependable adaptation in these systems.

RQ2 (Efficiency). To answer this research question, we carried out several experiments measuring the overheads incurred by the various DECIDE stages, the time taken to reconfigure the system when a violation occurred and the efficiency of adaptation decisions.

First, we monitored and analysed the CPU and communication overheads incurred by the DECIDE stages 1–4. As shown in Table 5.9, the CPU overheads for the RQV-based local capability analysis, the RQV-based local control loop and the MCKP knapsack problem solving in CLA selection, i.e., DECIDE stages 1, 3, and 4, respectively, are all negligible at under 40ms each, or below 0.4% when the local control loop is executed every 10s. The communication overheads incurred for sending and receiving a capability summary, i.e., DECIDE stages 1 and 2, is 71 bytes per peer UUV per major change[3]. This overhead is very low too, even for a typical inter-UUV bandwidth of 2.5Kbps [177].

Second, in all the experiments, the system recovered after sensor failures, performance drops and complete UUV failures within 800ms from the moment a UUV started executing the last periodic local control loop, for a typical inter-UUV bandwidth of 2.5Kbps. Hence, if the local control loop runs every 5s, the time to recovery was below 5.8s. This provides evidence that DECIDE can restore compliance with system-level and component-specific requirements very efficiently, and in fact, it required a fraction of the time taken by a monolithic approach (see Figure 5.9 later in this section).

Finally, we compared the number of measurements taken and the energy consumed

[3] Note that, the upper bound size in bytes of a capability summary transmitted by the i-th UUV is $\text{Comms\_Size} = N_i \times (m+1) \times \max_{1 \leq k \leq N_i \wedge 1 \leq j \leq m+1} \text{getChars}(\alpha_{ij}^k)$ where N_i is the number of configuration subsets, $m+1$ is the number of system-level QoS requirements (including system-level cost), and $\text{getChars}(\alpha_{ij}^k)$ is a function that gives the size in bytes of an RQV-based verification result for the k-th configuration and j-th system-level QoS requirement.

Table 5.9: Mean CPU and communication overheads for a three-UUV mission

DECIDE stage	CPU use [ms]	bytes sent	bytes received
1	38.3	71*	0
2	∼0	0	71*
3	0.7	0	0
4	25.6	0	0

* per peer UUV

Table 5.10: Comparison of DECIDE with the "ideal" system **

confidence level α	additional energy use	additional measurements
0.90	+18.26%	+12.54%
0.95	+18.30%	+12.58%
0.99	+20.62%	+9.97%

** averaged over 10 experiments

by the three-UUV DECIDE system with the values of the same metrics for an "ideal" system. In this "ideal" system (i) the sensor rates never varied from their nominal values; (ii) the globally optimal set of sensors satisfying requirements R1–R6 were used at all times; and (iii) all UUVs travelled with the minimum speed of 1m/s, to maximise the fraction of measurements that were accurate. This "ideal" system cannot be implemented in practice[4], but has the useful property that any practical system will use more measurements and more energy than it does. The results in Table 5.10 show the excess energy consumed and additional measurements taken by DECIDE, averaged over 10 experiments. Accordingly, DECIDE successfully decentralised the control loop of the UUV system with a modest loss in efficiency. In particular, DECIDE required at most 21% more energy and performed not more than 13% additional measurements, compared to the "ideal" solution.

RQ3 (Confidence level α). To examine the extent to which the confidence level α affects the adaptation behaviour of DECIDE, we carried out two sets of experiments with different α-confidence levels. In the first set of experiments, we compared the energy used by the three-UUV system using DECIDE and confidence value $\alpha \in \{0.90, 0.95, 0.99\}$ with the energy consumed by an "ideal" system (cf. previous research question). As shown in Table 5.10, higher confidence levels make the component capability summaries (5.7) more conservative, at the expense of increased system-level cost (e.g., the energy use for our system), and vice-versa. In the most conservative case however, i.e., when $\alpha = 0.99$, the energy used by DECIDE exceeds just by 20% the

[4]The inter-UUV bandwith does not permit the UUVs to exchange state information continuously as required by the ideal scenario.

"ideal" energy consumption.

An interesting observation from Table 5.10, concerns the number of additional measurements associated with the various α-confidence levels. Although the excess in energy consumption increases as the α-confidence level becomes larger, this is not the case with the excess in sensor measurements. This occurs because of two reasons. First, the capability summary sent by a component to its peers (5.7) is a discrete selected subset of the possible configurations of the component. Second, the system-level cost (i.e., QoS requirement R3) refers to the energy consumption of the multi-UUV system. Therefore, this is the objective that DECIDE aims to minimise when establishing the components' CLAs in (5.9).

In the second set of experiments, we used the same α confidence values and seeded the runtime behaviour of the three-UUV system with several failure patterns in which the degradation of service varied between small changes (10%) and more significant changes (70%) of the nominal sensor rates. We monitored the frequency with which these changes invalidate the CLAs determined by DECIDE, forcing the UUVs to carry out local analysis, to share their new capability summaries and to establish new CLAs (i.e., DECIDE stages 1–3). When using 90%, 95%, and 99% α confidence levels, the three-UUV system was able to handle these changes locally and to find a configuration that satisfies its UUV CLAs, 52.4%, 80%, and 83.8% of the times, respectively.

These results indicate that lower confidence levels lead system components to operate close to their optimal capabilities but make the system susceptible to smaller changes and the UUVs need to re-establish their CLAs with higher frequency. On the contrary, higher confidence levels make the system to operate more conservatively. This increases the system-level cost (e.g., the energy use for our system), but reduces the number of changes that cannot be handled by the local control loop, as a larger fraction of α of the control loop checks will find the component operating in an environment state that can be accommodated through local adaptation. For the three-UUV system, a 95% confidence level seems an acceptable trade-off between the increase in system-level cost and the ability of the system to handle changes and adapt locally.

RQ4 (Scalability). We carried out a set of experiments by varying the number of UUVs, in order to investigate how well DECIDE can scale with systems of different sizes. We experimented with systems comprising up to 32 UUVs (each equipped with three sensors) and compared the CPU time taken by each DECIDE stage and by the adaptation process for a variant of the multi-UUV system running a centralised RQV control loop.

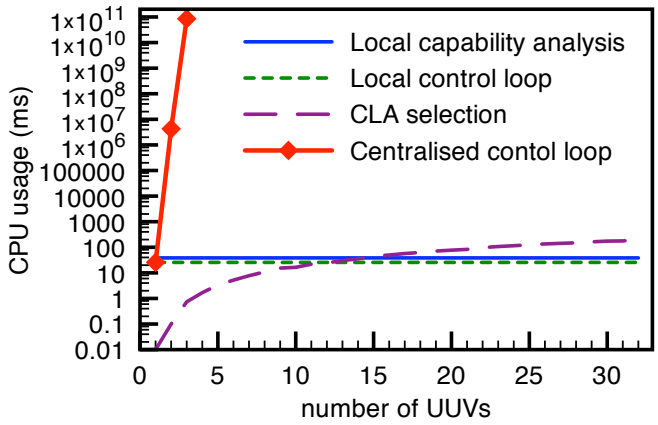


Figure 5.9: DECIDE scalability analysis.

As shown in Figure 5.9, the two RQV-based DECIDE stages (i.e., the local capability analysis and the local control loop) use the same small amount of CPU time irrespective of the size of the n-UUV system. The $O(n^2)$ CPU time taken by the CLA selection stage stays below 200ms for systems of up to 32 UUVs. In contrast, using a centralised control loop that applies RQV to the entire system model $M_1 \parallel M_2 \parallel \ldots \parallel M_n$ takes over 4200s for $n = 2$ and is unfeasible for $n > 3$. The CPU time shown in Figure 5.9 for the RQV of a complete model of a three-UUV system (i.e., 983.5 days) is an estimate we obtained based on the average verification time over a small subset of representative configurations from the configuration set that would need to be verified by this control loop.

Considering these results, we can safely state that DECIDE requires a fraction of the CPU time consumed by a centralised RQV control loop. Furthermore, the number of components comprising a distributed self-adaptive system has limited effect on the overheads incurred by DECIDE.

5.3.4 Threats to Validity

We identified several construct, internal and external threats that can influence the validity of the results presented in this chapter.

Construct validity threats are associated with the assumptions made when implementing the multi-UUV case study, and in the development of the stochastic models and QoS requirements for the system. To mitigate this threat, the QoS requirements R1–R6 from Tables 5.1 and 5.2 and the stochastic model are based on a validated case study from our previous work using a single UUV (Section 3.2). Furthermore, we

implemented the multi-UUV system simulator (Section 5.2]) using the well-established UUV platform MOOS-IvP [20].

Internal validity threats can originate from how the experiments were performed, and any bias introduced when obtaining and analysing the results. To reduce these threats, we examined a wide range of scenarios with various characteristics in terms of number of UUVs, nominal sensors rate and α-confidence level (cf. Table 5.7). Moreover, the failure patterns used in these scenarios consisted of different types of failures, including complete sensor failures and both minor and significant performance degradation of UUV sensors. Finally, the reported results were collected over multiple independent runs.

External validity threats can arise because of the difficulties to cast the QoS attributes and requirements of other distributed self-adaptive systems into the pattern given by (5.1)–(5.3) in Section 5.1.2. To limit this threat, we distilled this pattern from the growing body of research on RQV-driven self-adaptation in service-based, cloud-deployed and embedded software systems [32, 33, 35, 38, 68, 76, 94, 130]. Second, for other systems it may not be possible to identify α-confidence subsets of environment states. This threat is mitigated by the fact that DECIDE can operate with approximations of such subsets, which impact only the frequency of major changes. Finally, major changes may occur too frequently, leading to unacceptable overheads and "jitter" in component reconfigurations. DECIDE can alleviate this by increasing the α-confidence level (i.e., being more conservative), but our approach is not intended for systems with a high churn rate. However, we are aware that we evaluated DECIDE in a single case study. Therefore, evaluating DECIDE in other types of distributed self-adaptive software systems is necessary in order to confirm the generality of the approach.

5.4 Related Work

The work we introduce in this chapter touches both areas of decentralised control self-adaptive systems and runtime quantitative verification. In the following paragraphs we summarise the most relevant related work from these areas.

Decentralised-Control Self-Adaptive Systems. The subject of decentralised control in software systems has been studied to a great extent by various research communities, including the software engineering, intelligent systems, robotics, autonomous and adaptive systems communities. In particular, decentralised control in self-adaptive software systems has been developed using many approaches, as for example, agent-

based [210, 211] and service-based systems [62]. Wooldridge and Jennings [211, p. 116–118] define an agent-based system as a software system comprising one or multiple agents that can: (i) act autonomously with limited human interference; (ii) react to changes in the environment making decisions that allow the system to comply with design-time requirements; (iii) act pro-actively, if necessary; and (iv) communicate with other agents and take decisions to satisfy their design-time requirements. Likewise, a service-based system can dynamically recompose itself by discovering and selecting on-the-fly services in order to respond to evolving requirements, changes in its environment, and performance degradation of component services [62].

Tesauro and his colleagues [52, 197] introduced Unity, a decentralised architecture for self-managing distributed computing systems that enables self-configuration at initialisation and self-optimisation at runtime. Each system component in Unity is an autonomic element capable of controlling its own resources to meet its individual requirements as well as delivering services to other autonomic elements. A resource arbiter maintains a global view of resources demand–availability and is responsible to allocating the resources to the autonomic elements after evaluating a service-level utility function.

In [93], the authors present an approach for self-organising distributed systems in which each component maintains a consistent global configuration view using a reliable, totally ordered broadcast mechanism. After changes in the system, a configuration manager is responsible for adapting the affected component, provided that user architectural constraints are satisfied. Sykes et al. [195] propose the use of gossiping among system components as a solution to the scalability problems faced by [93]. Each component keeps a partial view of the system and by making use of a gossip protocol, the components can agree the adaptation plan (which is identical to a centrally-derived configuration) in logarithmic time with respect to the system size. A gossip-based protocol is also used in [101] to achieve decentralised and dynamic self-assembly of distributed services that can satisfy global functional and non-functional requirements.

Decentralised approaches for self-managing agent organisations where the agents can form coalitions dynamically in order to achieve a global objective are proposed in [142] and [207]. In [142], each member in the organisation uses only locally stored information and periodically contacts a subset of its peers in order to determine the necessary actions to improve the organisation's performance. In MACODO [207], agents are organised following a master-slave arrangement, with each master controlling a different organisation. When necessary, masters can exchange limited information and cooperate to form coalitions and achieve system-level objectives. This is similar to DECIDE components sharing their capability summaries in order to select their CLAs.

In [163], the authors introduce a decentralised market-based mechanism for enabling cloud-based service-oriented applications to adapt to changing QoS requirements. They consider the cloud as a marketplace where providers can offer their services and the system, using a double-action mechanism, can select dynamically from the pool of services currently available, those services that satisfy the system QoS requirements and optimise a utility function. Along the same line, [71] proposes a decentralised market-based and reputation-guided approach that attempts to minimise QoS violations in cloud applications by dynamic provisioning of cloud resources.

Despite the promising results presented by these approaches, they take adaptation decisions using simple heuristics (e.g., gossip-style [101, 195] and market-based [71, 163] heuristics), or bio-inspired optimisation techniques (e.g., ant and holonic systems [61] and reinforcement learning [176, 215]). These approaches cannot provide the strong guarantees required in mission-critical and business-critical application domains, as for example, in telehealth service-based systems [33], dynamic power management [38], and unmanned underwater vehicles [94]. DECIDE overcomes these issues and is able to guarantee that the decentralised-control system meets its QoS requirements in the presence of changes, recovering from failures whenever feasible.

Clearly, assuring predictable system operation in distributed-self adaptive systems is an important, but relatively new, area of research. To this end, recent approaches explored the use of formal methods to guarantee that decentralised-control self-adaptive systems meet their functional requirements [63, 80, 167]. Our work complements this research, since DECIDE focuses on the QoS requirements of distributed self-adaptive systems.

Runtime Quantitative Verification. Recent approaches to improve RQV overheads and extend the applicability of the technique to more complex software systems are extensively reviewed in Section 2.2.3. In Chapter 3 we proposed a set of conventional software engineering techniques (i.e., caching, limited lookahead and nearly-optimal reconfiguration) to reduce RQV overheads. Moreover, our EvoChecker approach, introduced in Chapter 4, shows that search-based techniques can improve further RQV efficiency, especially in software systems with large configuration space sizes.

DECIDE is in line with these approaches as it aims to reduce the RQV control loop overheads. Our approach can be considered both *incremental* and *compositional*; it is incremental as it reverifies only the impacted by the changes parts of a system; and it is compositional as it is capable of providing guarantees for system-level compliance with QoS requirements without generating the complete, monolithic model of the entire

system. Furthermore, DECIDE is a generic framework that supports a wide range of Markov models and system properties, whereas the aforementioned approaches are applicable only in specific variants of Markov models and/or system properties (cf. Table 2.3). Additionally, [44, 76, 94, 110, 154] are not capable of handling changes in the structure of the model, as DECIDE does. Hence, in case of structural changes the entire process needs to be repeated from the beginning. Compared to [130], DECIDE generates the CLAs on-thy-fly and without any external coordination, whereas [130] builds upon the assume-guarantee framework in which the assumptions are generated manually (although in [72] an effort is made to use machine learning for creating the assumptions).

Additionally, these approaches use centralised control loops, which restricts the on-the-fly use of RQV to small system models that can be analysed fast and verified with acceptable overheads at runtime. DECIDE tackles this challenge since all RQV-based stages within our approach analyse component models. Hence, DECIDE extends the applicability of RQV to large component-based models (cf. the scalability results in Figure 5.9). DECIDE addresses also the single point of failure introduced by centralised control loops because the control loop of a DECIDE component continues to operate even if a peer component and its control loop fail.

To the best of our knowledge, DECIDE is the first approach that employs RQV-based decentralised control loops for providing guarantees regarding the compliance of a distributed self-adaptive system with its QoS requirements.

5.5 Summary

In this chapter, we presented DECIDE, an approach that enables the engineering of distributed self-adaptive software systems with RQV-driven decentralised control loops. To this end, we developed a theoretical framework for the decentralisation of control loops in this type of systems. This new theoretical framework comprises the following stages: (1) RQV-driven local capability analysis for calculating component QoS capability summaries; (2) receipt of peer QoS capability summaries; (3) decentralised selection of component contribution-level agreements (CLAs) which assures that system-level QoS requirements are met; and (4) RQV-based local control loop that guarantees compliance of each component with its CLA and local requirements. DECIDE is also able to identify when a component is unable to satisfy its CLA through local adaptation. In this situation a major change occurred and DECIDE triggers the re-execution of stages (1)–(4) to establish new CLAs and restore compliance with system-level QoS

requirements.

We validated our approach and showed its effectiveness using a simulated distributed embedded system from the unmanned underwater vehicle domain. More specifically, DECIDE outperforms the current use of centralised RQV-based control loops for providing assurances in self-adaptive systems (Section 5.4) in several aspects. First, DECIDE is able to drive reconfiguration with CPU, energy, and memory consumption overheads that are several orders of magnitude lower. When compared to an "ideal", but practically infeasible variant of the self-adaptive system, our approach incurs only a modest increase in system-level costs (18–21% in our case study). Second, DECIDE can scale without any significant increase in overheads with systems of much larger sizes (up to 32 in our case study). Finally, distributed self-adaptive systems using DECIDE are not susceptible to the single point of failure (faced by centralised control loops) as system components can continue operating when a peer component fails completely, irrespective of which component this is.

Chapter 6

Engineering Trustworthy Self-Adaptive Systems

Self-adaptive software systems deployed in mission-critical and safety-critical application domains, e.g., healthcare, transportation, and finance, are expected to reconfigure themselves in response to changing environments, evolving requirements and unexpected system failures. In the previous chapters, we demonstrated how RQV can support self-adaptation in these systems and how the verification results generated by the technique can be used as evidence to ascertain the correctness of adaptation decisions. We also proposed variants of the technique that improve its performance (Chapters 3–4) and extend its applicability to distributed self-adaptive systems (Chapter 5).

Despite the extensive recent research in engineering self-adaptive systems [127, 143, 183], *assurances* is an aspect of this process that is still underexplored. Assurances are defined as the provision of evidence that a software system complies with its functional and non-functional requirements throughout the system's lifetime [45]. In conventional (non self-adaptive) systems such evidence can be obtained at design-time, i.e., during requirements engineering, system design or implementation. In self-adaptive systems, however, the high levels of uncertainty during system operation change completely the setting. Thus, assurances for self-adaptive systems entail not only providing evidence for requirement compliance at design-time, but also supplying new evidence at runtime, once an adaptation is performed, to confirm that compliance still holds [205].

The need for assurances was brought into the forefront of self-adaptive systems only recently [55, 56]. Existing research provides correctness evidence for specific aspects of the self-adaptive software, but does not consider a picture of the entire system. The

assurances are, thus, incomplete, since this correctness evidence is only one component of the established industry process for the assurance of safety-critical and mission-critical systems [26, 199]. In systems from these domains, assuring a software system entails the establishment of an *assurance case*, which standards such as [200] define as *"a structured argument, supported by a body of evidence, that provides compelling, comprehensible and valid case that a system is safe for a given application in a given environment"*.

Driven by these observations, in this chapter we bridge the gap between the established industry process and the current research on assurances for self-adaptive systems. To this end, we introduce ENTRUST, a tool-supported methodology for the ENgineering of TRUstworthy Self-adaptive sofTware systems[1]. The ENTRUST methodology spans both design-time and runtime activities, and integrates work on developing formally verified control loops [128], our work on runtime quantitative verification, and Goal Structuring Notation (GSN), an industry adopted standard for the formalisation of assurance arguments [105, 192]. ENTRUST uses formally verifiable models (e.g., timed automata) that conform to the MAPE workflow (Figure 2.4) for the development of the controller of the self-adaptive system. Another set of parametric stochastic models captures system and environmental uncertainty. The controller models are verified to produce evidence confirming the controller correctness, and this evidence is used for the partial instantiation of an assurance argument. The two sets of models are then combined with reusable ENTRUST components, application-specific sensors and effectors, and the managed software system to produce a self-adaptive system that is ready for deployment. Any missing evidence from the assurance argument is continually generated while the self-adaptive system is running. This new evidence is used to incrementally update the assurance argument of the ENTRUST system. To the best of our knowledge, ENTRUST is the first fully-fledged methodology for the engineering of trustworthy self-adaptive software systems and their associated assurance cases.

The main contribution of this chapter is the ENTRUST methodology for engineering trustworthy self-adaptive systems, supported by formal assurance cases. We also propose a formally verifiable controller architecture for self-adaptive systems that integrates RQV into controller modules, and a set of controller properties that ENTRUST controllers must satisfy. We describe these contributions in Section 6.1. We present the ENTRUST implementation in Section 6.2. In Section 6.3, we report results from evaluating ENTRUST to developing self-adaptive systems across application domains. We review related work in Section 6.4 and summarise ENTRUST in Section 6.5.

[1]ENTRUST is a joint work with Prof. Tim Kelly and Dr. Ibrahim Habli from University of York, UK and with Prof. Danny Weyns and Usman Iftikhar from Linnaeus University, Sweden.

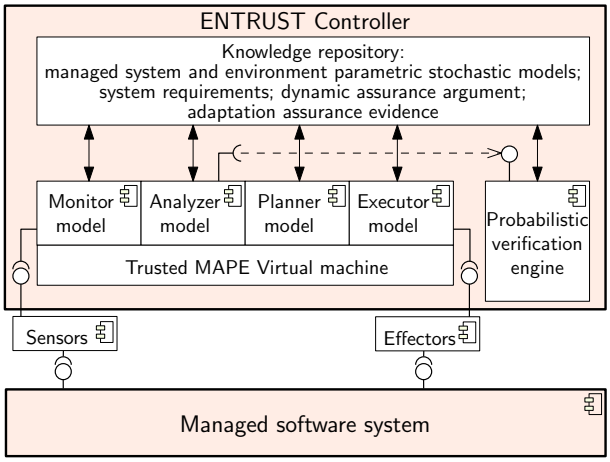


Figure 6.1: Architecture of an ENTRUST self-adaptive system.

6.1 ENTRUST Methodology

The high-level architecture of an ENTRUST self-adaptive system is based on the MAPE control loop (Figure 2.4) and is depicted in Figure 6.1. The self-adaptive system comprises an external *ENTRUST controller*, a managed software system and their communication mechanisms *Sensors* and *Effectors*. As always in this thesis, we assume that the managed software system is already available and its components can execute low-level commands required for self-adaptation. Thus, the focus of the ENTRUST methodology is on developing the controller and providing assurances for its correct implementation at design-time and reliable operation at runtime. The controller comprises: i) a set of formally verifiable controller models that correspond to the monitor-analyse-plan-execute steps of the MAPE loop; ii) a knowledge repository that contains the system requirements to be assured at runtime, the parametric stochastic models of the system and its environment, the adaptation assurance evidence and the updated assurance argument; and iii) the reusable components of our controller, i.e., a trusted MAPE virtual machine (which is developed by our collaborators Prof. Danny Weyns and Usman Iftikhar) [128] and a probabilistic verification engine (e.g., verification libraries of the probabilistic model checker PRISM [149]).

The controller models form an application-specific network of interacting timed automata [7], specified in the modelling language of UPPAAL [18]. At design-time these models are verified to establish key controller correctness properties, including reachability and liveness. At runtime, the virtual machine directly interprets and executes these models. The use of this model-driven engineering approach is a major benefit

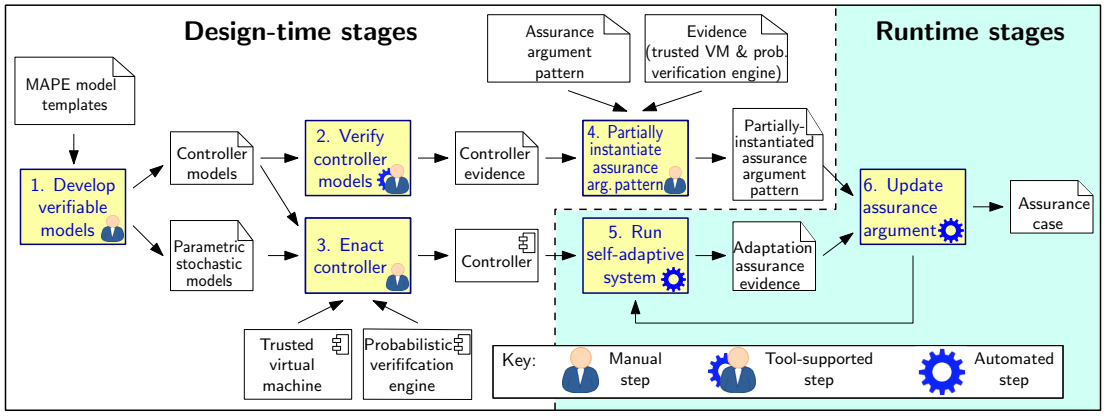


Figure 6.2: ENTRUST self-adaptive system and assurance case methodology.

since it does not require any model-to-text transformation of the verified models into executable code, which is a complex, and potentially error-prone operation. However, providing assurance evidence for the correct execution of the controller models by the virtual machine is crucial. This evidence, as we discuss later in Section 6.1.4, is provided by our collaborators when they developed and verified the virtual machine [128].

The execution of the controller models follows the typical execution of the MAPE loop (cf. Section 2.2.1). The *monitor model* collects information about the system and its environment through sensors. Elements of this information that are relevant to the behaviour of the system and its environment, and which correspond to the unknowns in the parametric stochastic models, are used to produce concrete instances of stochastic models. The *analyzer model* reverifies the compliance of the self-adaptive system with its requirements. This continual verification is carried out using the *probabilistic verification engine*. In case of a requirement violation, the analyzer uses the verification results to determine a new system configuration that restores this compliance. The *planner model* is responsible for deriving a stepwise plan to realise the new configuration, and this plan is implemented by the *executor model* through *effectors*.

The ENTRUST methodology for developing high assurance self-adaptive systems and their corresponding assurance cases consists of four design-time and two runtime stages (Figure 6.2). In Stage 1, the four controller models of the MAPE loop and the parametric stochastic models of the managed system and its environment are developed. The controller models are devised by specialising a set of ENTRUST controller model templates [99] (which contain elements of the MAPE loop common across applications and placeholders for application-specific functionality). In Stage 2, the controller models are verified using a trusted model checker to produce evidence that the controller

satisfies a set of generic controller properties, e.g., safety and liveness. An instance of the controller is generated in Stage 3 by integrating the controller and parametric stochastic models with reusable ENTRUST components (virtual machine and probabilistic verification engine) and application-specific sensors and effectors. In Stage 4, evidence regarding the controller correctness, generated in Stage 2, is combined with a generic assurance argument pattern to produce a partially instantiated assurance argument. This incomplete argument contains placeholders for the assurance evidence that can be obtained only after the system starts executing, i.e., after the uncertainties associated with the system and its environment are resolved.

At runtime, the deployed self-adaptive system executes the MAPE loop and dynamically reconfigures itself in response to environmental and internal changes (Stage 5). This reconfiguration produces adaptation assurance evidence that establishes both the correctness of the configuration results and the validity of the derived reconfiguration plan. Stage 6 uses this adaptation assurance evidence to fill in the placeholders from the *partially instantiated assurance argument*, and to derive the fully-fledged assurance argument of the system.

In the following sections we present the stages of the ENTRUST methodology in more detail. We use the self-adaptive UUV embedded system described in Section 2.2.1.1 and the QoS requirements from Table 6.1 as a running example. For each stage, we provide a general description and we illustrate the application of ENTRUST to instantiate the UUV controller and to generate the corresponding assurance argument.

Table 6.1: QoS requirements for the UUV self-adaptive system

ID	Informal description
R1	"The UUV should take at least 20 measurements of sufficient accuracy for every 10 metres of mission distance."
R2	"The energy consumption of the sensors should not exceed 120 Joules per 10 surveyed metres."
R3	"If requirements R1 and R2 are satisfied by multiple configurations, the UUV should use one of these configurations that minimises the cost function $cost = w_1 E + w_2 sp^{-1}$" ($w_1, w_2 > 0$ are weights, E is the energy consumed by sensors per 10 surveyed meters, and sp is the UUV speed)
R4	**(failsafe)** "If a suitable configuration is not identified within ten seconds after a sensor rate change, the UUV speed must be reduced to 0m/s. This ensures that the UUV does not advance more than a certain distance without taking appropriate measurements, and waits until the controller identifies a suitable configuration (e.g., after the UUV sensors recover) or new instructions are provided by a human operator."

6.1.1 Stage 1: Development of Verifiable Models

During the initial stage of the ENTRUST methodology, two sets of formally *verifiable* models are developed, i.e., controller models and parametric stochastic models. A trusted model checker can be used to analyse the controller models and verify their compliance with a set of key correctness properties (Section 6.1.2).

Controller models. The controller models carry out the tasks of the controller of the self-adaptive system. Their structure comprises an application-specific network of interacting timed automata that matches the four steps of the MAPE control loop. The development of these models is facilitated by specialising application-independent ENTRUST controller model templates adapted from the recent work of de La Iglesia and Weyns [99]. These templates are preconfigured with elements of the MAPE loop common in self-adaptive systems and leave application-specific aspects to be developed later in the form of placeholders. There are two types of MAPE model templates [99]:

i) *event triggered*, in which the monitor automaton is activated by a sensor-generated signal indicating a change in the managed system or its environment;

ii) *time triggered*, in which the monitor automaton is periodically activated by a an internal clock;

We extended these templates by adding elements specific to the ENTRUST controller (e.g., probabilistic verification engine) and the type of managed systems handled by ENTRUST (e.g., failsafe configuration). We define these templates in UPPAAL [18], since it is a well-regarded modelling language and a mature verification suite.

We depict the event-triggered automaton templates in Figure 6.3. States are annotated with atomic propositions that are true in those states, e.g., `ProcessSensorData`, `PlanCreated`. Transitions are annotated with boolean *guards* that must hold for the transition to take place, and actions are executed once transitions are taken. Two automata can synchronise their transitions (so that they are taken at the same time) through synchronisation channels, i.e., pairs comprising a '!'-decorated sent **signal!** and a '?'-decorated received **signal?** with the same name, e.g., **startAnalysis!** and **startAnalysis?** from the monitor and analyzer automata, respectively. Signals in angle brackets '⟨⟩' are placeholders for application-specific signal names. Finally, *guards* and actions decorated with brackets '()' represent application-specific C-style functions.

To specialise these templates for a self-adaptive system, software engineers need:

i) to replace the signal placeholders with real signal names;

ii) to define the guard and action functions;

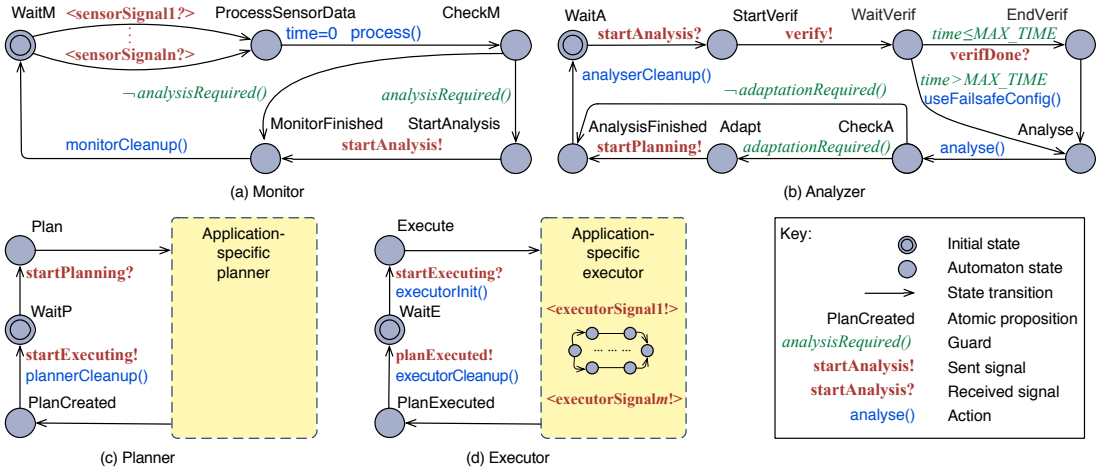


Figure 6.3: Event-triggered MAPE model templates.

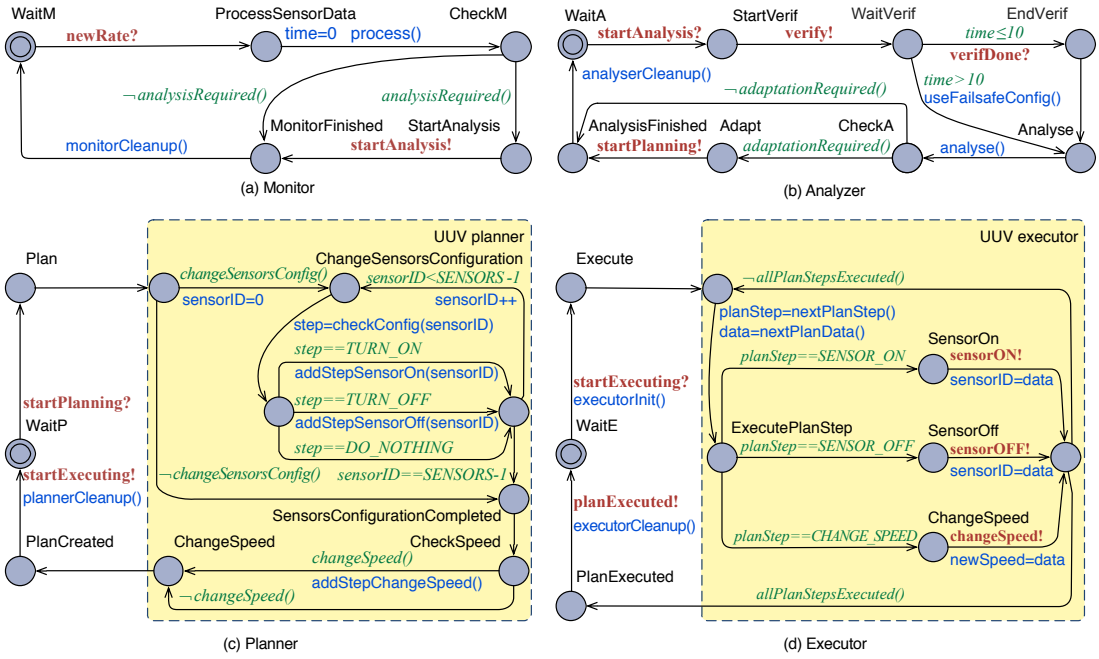


Figure 6.4: Instantiation of UUV MAPE automata based on the event-triggered ENTRUST model templates.

iii) to devise the (shaded) application-specific automaton regions from Figure 6.3.

In the monitor automaton, for instance, the engineers first need to replace the placeholders $\langle$**sensorSignal**$_1$**?**$\rangle$, ..., $\langle$**sensorSignal**$_n$**?**$\rangle$ with sensor signals announcing relevant changes in the managed system. They must then implement the functions process(), *analysisRequired()* and monitorCleanup(), whose roles are to process the sensor data, to decide if the change specified by this data requires the "invocation" of the

analyzer through the **startAnalysis!** signal, and to carry out any cleanup that may be required, respectively.

In the analyzer automaton, signals **verify!** and **verifDone?** are used to invoke the probabilistic verification engine and to receive notification once the engine finishes its execution, respectively. The engineers must then realise the functions useFailsafeConfig(), analyse() and *adaptationRequired()*, whose purpose are to force the system to use the failsafe configuration if a time threshold is exceeded (i.e., the guard $time > MAX\_TIME$ holds), to determine a configuration that satisfies the system QoS requirements by analysing the verification results, and to check if an adaptation is needed to switch to this configuration, respectively. If adaptation is required, the analyzer starts the planner automaton through the **startPlanning!** signal. The implementation of the analyserCleanup() function performs any necessary cleanup.

In the planner automaton, the yellow shaded region should be developed to construct a stepwise reconfiguration plan that switches the system to the new configuration upon receiving the **startPlanning?** signal. Also, the function plannerCleanup() should be implemented to carry out any required cleanup. The **startExecuting!** signal invokes the executor automaton.

In the executor automaton, the engineers should develop the yellow shaded automaton region in order to realise the reconfiguration plan. Likewise, the placeholders $\langle \textbf{executorSignal}_1\textbf{?} \rangle$, ..., $\langle \textbf{executorSignal}_n\textbf{?} \rangle$ should be replaced with appropriate executor signals implementing the adaptation decisions in the managed system. Finally, the functions executorInit() and executorCleanup() should be implemented in order to initialise the executor automaton and to make any necessary cleanup, respectively.

Example 6.1. We instantiated the ENTRUST model templates for the UUV system, obtaining the automata shown in Figure 6.4. The signal **newRate?** is the only sensor signal that the monitor automaton needs to deal with, by reading a new UUV-sensor measurement rate (in process()) and checking whether this rate has changed to such extent that a new analysis is required (in *analysisRequired()*). If analysis is required, the analyzer automaton sends a **verify!** signal to invoke the probabilistic verification engine, and thus verifies which UUV configurations satisfy requirements R1 and R2 and with what *cost*. The function analyse() uses the verification results to select a configuration that satisfies R1 and R2 with minimum *cost* (cf. requirement R3). If no such configuration exists or the time limit is exceeded (i.e., the guard $time > 10$ holds

and the useFailsafeConfig() function is executed), a zero-speed configuration is selected (cf. requirement R4). If the selected configuration is not the one in use, *adaptationRequired()* returns true and the **startPlanning!** signal is sent to initiate the execution of the planner automaton. The planner assembles a stepwise plan for changing to the new configuration by first switching on any UUV sensors that require activation, then switching off those that are no longer needed, and finally adjusting the UUV speed. These reconfiguration steps are carried out by the executor automaton using the **sensorON!**, **sensorOFF!** and **changeSpeed!** signals handled by the effectors from Figure 6.1.

Parametric stochastic models. The parametric stochastic models capture the behaviour of the managed system and the environment in which the system operates. The parameters represent aspects of the system that are unknown at design time and become known only through monitoring the managed system at runtime, e.g., the operating rate of the UUV sensors. These models are defined in the PRISM high-level modelling language [149]. We chose PRISM since it is an established suite for modelling and analysing stochastic systems and we have extensive experience with it. However, other stochastic modelling languages and tools (e.g., MRMC [133]) could be used. Thus, the models can be any of the supported PRISM models, including DTMCs (Section 2.1.1.1) and CTMCs (Section 2.1.1.2). Similarly, system QoS requirements are specified in the appropriate probabilistic temporal logic, e.g., PCTL (Section 2.1.2.1) and CSL (Section 2.1.2.2).

Example 6.2. The CTMC model M_i of the i-th UUV sensor is shown in Figure 2.3 and a description of its operation is given in Example 2.2. The model M of an n-sensor UUV is given by the parallel composition of the n sensor models: $M = M_1||...||M_n$. The QoS system requirements from Table 6.1 are specified using CSL as follows:

R1: $R^{\text{“measurement”}}_{\geq 20}[C^{\leq 10/sp}]$

R2: $R^{\text{“energy”}}_{\leq 120}[C^{10/sp}]$

R3: minimise $w_1 E + w_2 sp^{-1}$, $E = R^{\text{“energy”}}_{=?}[C^{10/sp}]$

where $10/sp$ is the time taken to travel 10m at speed sp.

The failsafe requirement **R4** encodes a condition-action policy [127], so it does not need to be specified using CSL.

6.1.2 Stage 2: Verification of Controller Models

During this ENTRUST stage, a trusted model checker is used to manually verify the controller models developed in the previous stage (Section 6.1.1). This operation provides assurance evidence that the controller satisfies key correctness properties including safety and liveness. A non-exhaustive list of such properties is shown in Table 6.2.

In order to carry out this verification and collect the assurance evidence, an end-to-end functional set of automata is required. This means realising the application-specific parts of the controller models (e.g., shaded regions in planner and executor automata). Additionally, automata that simulate the sensors, probabilistic verification engine and effectors from Figure 6.1 must be defined to enable this verification. The sensors, verification engine and effectors automata have to synchronise with the relevant monitor, analyzer and executor signals, respectively. For instance, the auxiliary sensor automaton must synchronise with the monitor automaton by sending the signal **newRate!**; this is the starting signal initiating the execution of the entire MAPE loop. The sensors and verification automata have to exercise all possible paths through the monitor, analyzer and planner automata (and indirectly the executor automaton). To this end, they can nondeterministically populate the knowledge repository with data that satisfies all the different guard combinations. Alternatively, a finite collection of the two automata can be used to verify subsets of all possible MAPE paths, as long as the union of all such subsets covers the entire behaviour space of the MAPE network of automata.

The time and resources required to verify the controller models depend both on the size of the automata (e.g., Figure 6.4) and the code complexity of the guard and action (C-style) functions. The larger the automata and the more complex the code, the higher the overheads for completing the verification. Thus, verifying complex controller models becomes challenging when the available resources are limited.

Example 6.3. We used the UPPAAL model checker [18] to verify that the network of MAPE automata of the UUV system (Figure 6.4) satisfies all the correctness properties from Table 6.2. To achieve this, we defined simple sensors, verification engine and effectors automata as shown in Figure 6.5. We used a simple one-state sensor automaton with transition returning to its single state for the outgoing signal **newRates!**. Similarly, the effector automaton had a single state and its transitions return to that state for each of the received signals **sensorON?**, **sensorOFF? changeSpeed?** and **planExecuted?**. The verification engine automaton simulates the probabilistic analysis of the

Table 6.2: Generic properties that should be satisfied by an ENTRUST controller

ID	Informal description	Specification in computation tree logic [49]
P1	The ENTRUST controller is deadlock free.	A□ not deadlock
P2	Whenever analysis is required, the Analyser eventually carries out this action.	A□ (Monitor.StartAnalysis → A◇ Analyzer.Analyse)
P3	Whenever the system requirements are violated, a stepwise reconfiguration plan is assembled.	A□ (Analyzer.Adapt → A◇ Planner.PlanCreated)
P4	Whenever a stepwise plan is assembled, the Executor eventually implements it.	A□ (Planner.PlanCreated → A◇ Executor.PlanExecuted)
P5	Whenever the Monitor starts processing the received data, it eventually terminates its execution.	A□ (Monitor.ProcessSensorData → A◇ Monitor.Finished)
P6	Whenever the Analyser begins the analysis, it eventually terminates its execution.	A□ (Analyzer.Analyse → A◇ Analyzer.AnalaysisFinished)
P7	A plan is eventually created, each time the Planner starts planning.	A□ (Planner.Plan → A◇ Planner.PlanCreated)
P8	Whenever the Executor starts executing a plan, the plan is eventually executed.	A□ (Executor.Execute → A◇ Executor.PlanExecuted)
P9	Whenever adaptation is required, the current configuration and the best configuration differ.	A□ (Analyzer.Adapt → currentConfig != newConfig)

stochastic system models by receiving the signal **verify?** and transmitting the signal **verifDone!** to the analyzer automaton. To exercise all possible paths of the MAPE automata from Figure 6.4 we defined a finite collection of sensor–verification engine automata pairs, instrumented with the appropriate inputs. The function prepareVerificationData() selects randomly verification data from this collection with the purpose to force the generation of all possible plans in the planner automaton. To this end, we assessed all possible outcomes of the *guards*, including those in the application-specific planner and executor automata (e.g., *changeSensorsConfig()*, *changeSpeed()*).

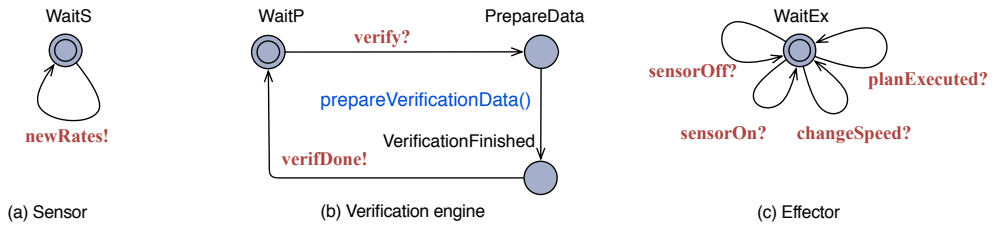


Figure 6.5: Auxiliary sensor, verification engine and effector automata used for verifying the generic controller properties from Table 6.2 for the UUV system.

6.1.3 Stage 3: Controller Enactment

This ENTRUST stage is responsible for assembling the controller of the self-adaptive system. First, the auxiliary sensor, verification engine and effector automata used in Stage 2 for verifying the generic controller properties in UPPAAL are replaced by application-specific components that carry out the designated functionality. In the current ENTRUST version we implemented abstract Java classes that provide this functionality, and which must be specialised for each application. Listing 6.1 shows the Effector abstract class. These abstract classes are available on our Github repository[2]. Thus, the specialised sensors and effectors must use the APIs of the managed software system to observe its state and environment, and to modify its configuration, respectively. The verification engine, which employs the verification libraries of the PRISM probabilistic model checker [149], must be specialised to perform two tasks. First, it has to instantiate the parametric stochastic models using the actual values of the managed system and environment parameters (provided by the sensors). Second, it has to verify the application-specific QoS requirements for alternative system configurations.

Once these application-specific components are implemented, they are integrated with the controller and stochastic models devised in Section 6.1.1 and with the trusted virtual machine. The final outcome is a functional controller, which will be then combined with the managed software system through an application-specific process to complete the engineering of the self-adaptive system.

Example 6.4. To assemble a fully-fledged ENTRUST controller for the UUV system, we implemented Java classes that extend the functionality of the abstract *Sensor*, *Effector* and *VerificationEngine* classes from the ENTRUST distribution. Listing 6.2 shows the instantiated effector class for the UUV system. The specialised sensors and effectors synchronise with the *monitor* and *executor* automata through the application-specific signals **newRate!** and **sensorOn?**, **sensorOff?**, **changeSpeed?**, **planExecuted?**, respectively. These classes invoke the relevant API methods of our UUV simulator (developed using the open source MOOS-IvP middleware [20]). The specialised verification engine synchronises with the relevant application-independent signals (**verify?** and **verifDone!**) from the *analyzer* automaton, instantiates the parametric sensor models M_i from Figure 2.3, $1 \leq i \leq n$, and verifies the CSL-encoded requirements from Example 6.2.

[2] `https://github.com/gerasimou/ENTRUST`

Listing 6.1: Effector abstract class.

```
1  package controller;
2
3  import java.util.HashMap;
4  import activforms.engine.ActivFORMSEngine;
5  import activforms.engine.Synchronizer;
6
7  public class Effector extends Synchronizer{
8
9    private ActivFORMSEngine vm;                  // trusted VM
10   private int executorSignal1, executorSignaln; // signal(s)
11   private Object comm;                          // comm handle
12
13   /** Constructor: create a new effector */
14   public Effector(ActivFORMSEngine vm, Object comm){
15     //assign handlers
16     this.vm                = vm;
17     this.comm              = comm;
18
19     //get signal(s) ID
20     executorSignal1   = vm.getChannel("executorSignal1");
21     ...
22     executorSignaln   = vm.getChannel("executorSignaln");
23
24     //register signals
25     vm.register(executorSignal1, this, "effectorData");
26     vm.register(executorSignaln, this, "effectorData");
27   }
28
29   /** Executed when receiving one of the registered signals.
30       Upon receiving such a signal, the Effector must realise
31       the appropriate action to the managed system */
32   @Override
33   public void receive(int channelID, HashMap<String, Object> data){
34     if (channelID == executorSignal1){
35       Object action = data.get("effectorData");
36       //realise adaptation action to the managed system
37       //TODO...
38     }
39     else if (channelID == executorSignaln){
40       Object action = data.get("effectorData");
41       //realise adaptation action to the managed system
42       //TODO...
43     }
44   }
45 }
46
```

Listing 6.2: Effector class for the UUV system.

```
1  package controller;
2
3  import java.util.HashMap;
4  import activforms.engine.ActivFORMSEngine;
5  import activforms.engine.Synchronizer;
6
7  public class Effector extends Synchronizer{
8
9    private ActivFORMSEngine vm;                                    // trusted VM
10   private int sensorOn, sensorOff, changeSpeed, planExecuted; // signal(s)
11   private PrintWriter comm;                                       // comm handle
12
13
14   /** Constructor: create a new effector */
15   public Effector(ActivFORMSEngine vm, Object comm){
16     //assign handlers
17     this.vm           = vm;
18     this.comm         = (PrintWriter)comm;
19
20     //get signal(s) ID
21     sensorOn          = vm.getChannel("sensorOn");
22     sensorOff         = vm.getChannel("sensorOff");
23     changeSpeed       = vm.getChannel("changeSpeed");
24     planExecuted      = vm.getChannel("planExecuted");
25
26     //register signals
27     vm.register(sensorOn, this, "sensorID", "currentConfig");
28     vm.register(sensorOff, this, "sensorID", "currentConfig");
29     vm.register(changeSpeed, this, "sensorID", "currentConfig");
30     vm.register(planExecuted, this, "sensorID", "currentConfig");
31   }
32
33   /** Executed when receiving one of the registered signals....*/
34   @Override
35   public void receive(int channelID, HashMap<String, Object> data){
36     if (channelID == sensorOn){
37       Object sensorData = data.get("sensorId");
38       comm.println(sensorData); comm.flash();
39     }
40     else if (channelID == sensorOff){
41       Object sensorData = data.get("sensorId");
42       comm.println(sensorData); comm.flash();
43     }
44     else if (channelID == changeSpeed){
45       double newSpeed = (Double) data.get("newSpeed");
46       comm.println(newSpeed); comm.flash();
47     }
48     else if (channelID == planExecuted){
49       //TODO: cleanup effector, if needed
50     }
51   }
52 }
53
```

6.1.4 Stage 4: Partial Instantiation of Assurance Argument Pattern

During this ENTRUST stage, engineers use assurance evidence generated in the previous stages of our methodology and packaged with the external components used by ENTRUST to partially instantiate an *assurance argument pattern*, specifically developed for self-adaptive software systems. The evidence includes the controller assurance evidence produced in Stage 2 of the ENTRUST methodology, as well as testing evidence[3] regarding the correctness of the trusted virtual machine and the probabilistic model checker PRISM [149]. The outcome of this process is an incomplete (partially-developed) assurance argument. This argument provides as much evidence as available at design-time and leaves placeholders for evidence that can only be obtained at runtime, when the unknowns associated with the self-adaptive system are resolved.

In this work, we use the Goal Structuring Notation (GSN) [138] (a graphical argument notation) for constructing the assurance arguments. GSN is a community standard [105] and has been adopted by many companies operating in safety-critical domains, e.g., aerospace, defence, and healthcare. Recent examples include its use for structuring the assurance argument for the wheel braking software system of an aircraft [122] and the control software for a prototype autonomous vehicle [119]. We chose GSN due to its wide use in industry for establishing the assurance of software systems. It is also the argument notation with which our collaborators are most familiar with[4].

Figure 6.6 shows the basic elements of GSN [138]. These elements can be used to construct an assurance argument (also termed a "goal structure") by showing how assurance-related claims (termed *goals*) are decomposed into sub-claims using *strategies* until assurance evidence (*solutions*) can be provided to support these (sub-)claims. These elements can be linked together using the *supported by* linkage or its extensions *multiplicity* and *optionality*, which represent 'zero or more' and 'zero or one' relationships between GSN elements, respectively. When specifying a claim or strategy it might be useful to annotate (using the *in context of* linkage) the *context* in which it should be interpreted, any relevant *assumptions*, or any *justifications* required to explain why the claim or strategy is included in the assurance argument. Some goals or strategies can be left *undeveloped*, *uninstantiated* or satisfied through several alternatives (*choice*). An *AwayGoal* is a modular element of GSN that partitions the argument into distinct, but interrelated modules of arguments. Figure 6.7 depicts an example assurance argument.

[3] Evidence for the virtual machine and PRISM can be found at `https://people.cs.kuleuven.be/~danny.weyns/software/ActivFORMS` and `https://github.com/prismmodelchecker`, respectively.

[4] Recall that ENTRUST is a joint work with Prof. Tim Kelly and Dr. Ibrahim Habli from University of York, UK and with Prof. Danny Weyns and Usman Iftikhar from Linnaeus University, Sweden.

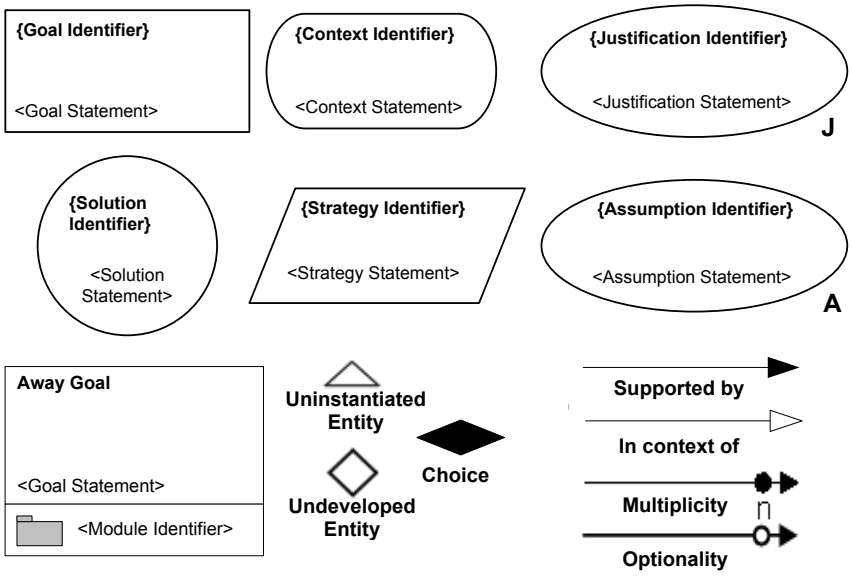


Figure 6.6: Main GSN elements for constructing an assurance argument.

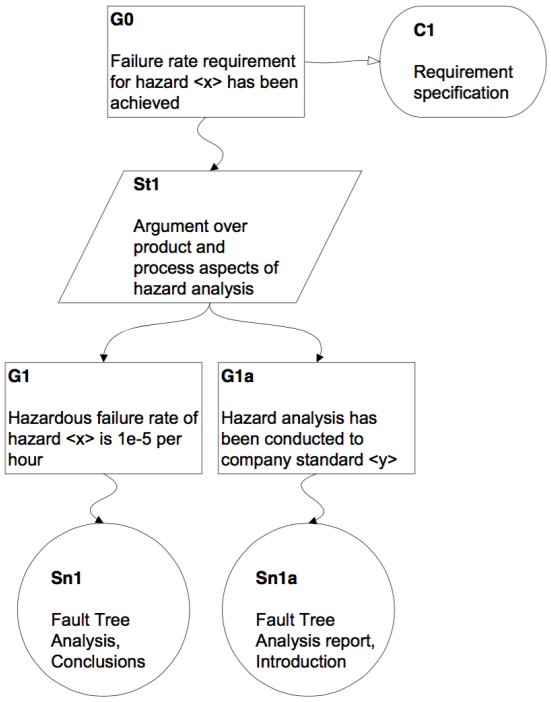


Figure 6.7: An example GSN assurance argument taken from [105] that shows how goal *G0* is decomposed into goals *G1* and *G1a* using strategy *St1*, and how evidence *Sn1* and *Sn1a* is used to establish the truth of the statements in the goals *G1* and *G1a*, respectively.

ENTRUST assurance argument pattern

The ENTRUST *assurance argument pattern* (Figure 6.8) builds on previous work of our collaborators on software safety assurance [120, 121]. Based on this work, a catalog of reusable assurance argument patterns is proposed in [119]. Each pattern considers the contribution made by the software to system hazards for a particular class of systems and scenarios. The assurance case reasoning is then structured based on the refinement of software requirements through the various stages of the system's lifecycle, including requirements engineering, design, implementation, and testing. Nevertheless, system and environment uncertainty affecting self-adaptive software systems means that, for this class of systems, the refinement process cannot be completed at design-time (i.e., before the system starts executing). For self-adaptive systems, this refinement is a continual process where different design features and code elements are dynamically reconfigured and executed during self-adaptation [59]. Therefore, claims and supporting evidence for meeting the application-specific requirements must vary with self-adaptation, and thus ENTRUST assurance arguments must evolve dynamically at runtime.

In the ENTRUST pattern, the *ReqsSatisfied* goal states that the application-specific requirements are always satisfied. Note that in the context of ENTRUST this assurance pattern concerns only the self-adaptation elements of the system (which conforms to the architecture from Figure 6.1). The justification of the derivation, validity and completeness of these application-specific requirements are addressed as part of the overall system assurance argument (which is outside the scope of the software assurance argument). *ReqSatisfied* can be supported by a pair of goals, *ReqsConfiguration* and *Reconfig*, stating that the system satisfies the application-specific requirements through the current configuration or through a reconfiguration, respectively. That is, the pattern shows that we are either guaranteeing that the current configuration n (specified in the context *ConfigDef*) satisfies the requirements or that the ENTRUST controller will plan and execute a reconfiguration that will satisfy these requirements (specified in the justification *Reconfig*).

The pattern uses the strategy *ConfigProps* to indicate how the system requirements are refined into, and covered by, the Critical Properties (*CPs*). This is specified by the away goal *CPsIdentify* (Figure 6.9) which is based on the Identification Software Safety Argument pattern defined in the existing GSN pattern catalogue [119]. The strategy *ConfigProps* also justifies how the goal *ReqsConfiguration* is met by the sub-goals *CPx_ Achieved* which signify that each of the critical properties are achieved when the system executes the configuration n. For each goal *CPx_ Achieved*, the pattern provides

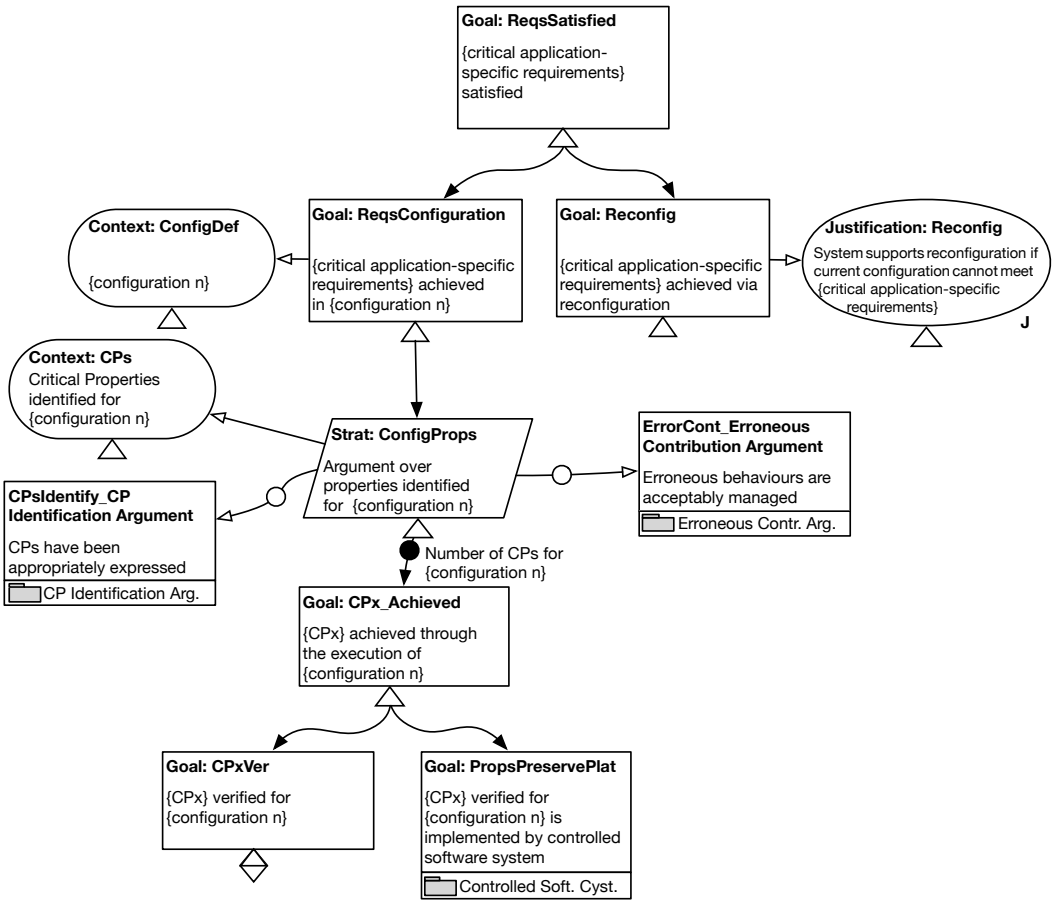


Figure 6.8: ENTRUST assurance argument pattern.

the goals *CPxVer* and *PropsPreservedPlat* which refer to its verification and implementation by the managed software system, respectively. Finally, a new configuration has the potential to introduce erroneous behaviours (e.g., deadlocks and stack overflows). The justification for the absence of these errors is provided via the away goal *ErrorCont* (Figure 6.10), which is based on the Hazardous Contribution Software Safety Argument pattern, also available in the catalogue [119].

The partial instantiation of the assurance argument pattern in the last design-time stage of ENTRUST produces a *partially-developed* assurance argument [59]. The argument includes evidence that is available at design-time, i.e., that system requirements are appropriately expressed using critical properties (*CPsIdentify*), that the reusable components virtual machine and probabilistic verification engine are correct and do not introduce any errors to the system (*ErrorCont*), and that the controller models satisfy key correctness properties. Uninstantiated solutions (e.g., CP_R1Res from the

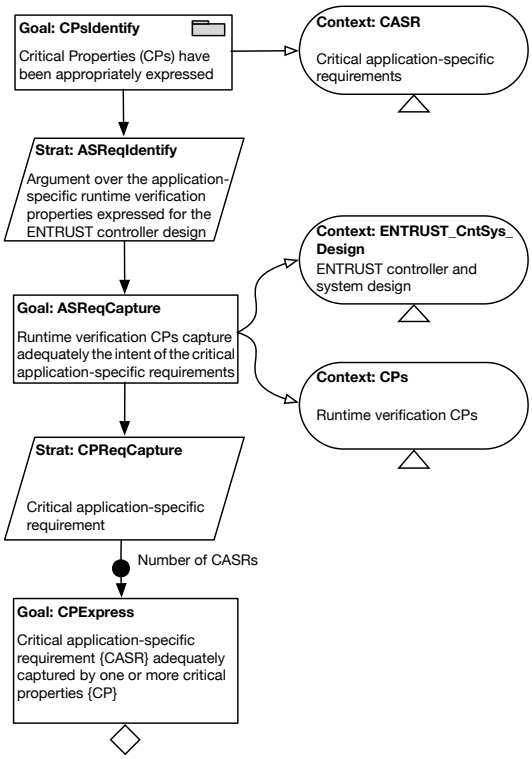


Figure 6.9: Away goal CPsIdentify which shows how application-specific requirements are captured by one or more critical properties.

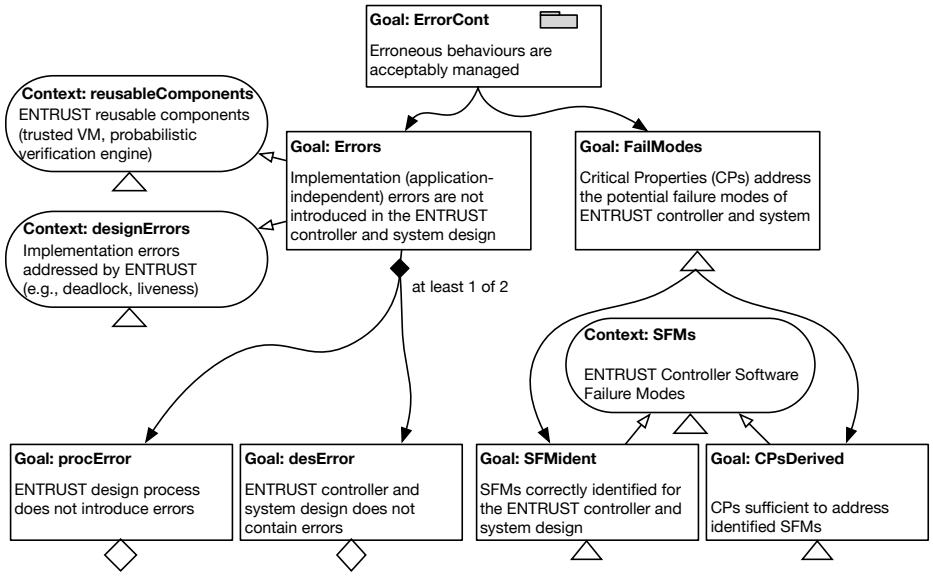


Figure 6.10: Away goal ErrorCont which indicates that (i) the design process of ENTRUST, and the reusable components virtual machine and probabilistic verification engine do not introduce any errors; and (ii) the identified critical properties address any failures of the ENTRUST self-adaptive system.

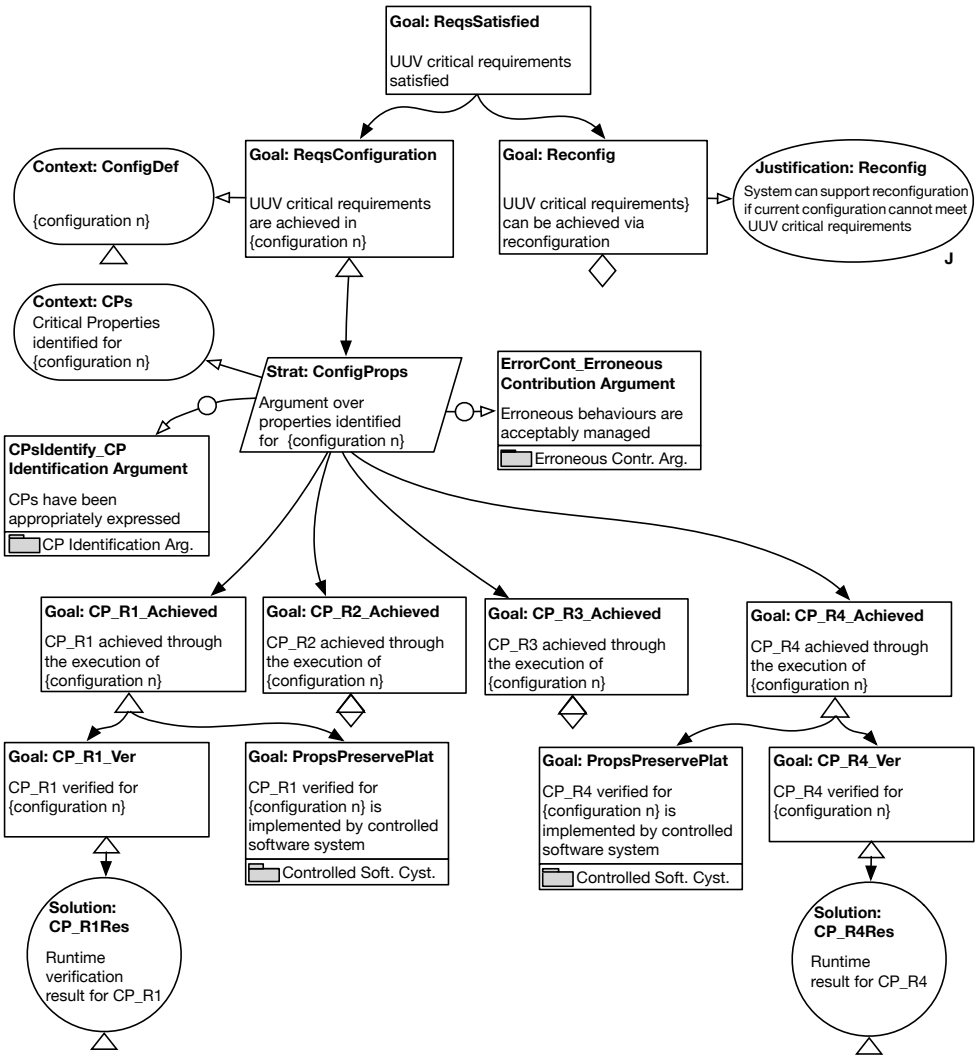


Figure 6.11: Partially-instantiated assurance argument for the UUV system.

partially-instantiated assurance argument pattern for the self-adaptive UUV system in Figure 6.11) form the placeholders for evidence that can be produced only based on operational data while the system is running, i.e., runtime verification evidence supporting the critical properties *CPs* of an active configuration.

Example 6.5. Figure 6.11 shows the partially-instantiated assurance argument pattern for the self-adaptive UUV system. For clarity reasons, we only show the expansion for requirement R1 and failsafe requirement R4. We leave R2 and R3 undeveloped; the GSN elements for these two requirements are entirely similar to those for R1. Crit-

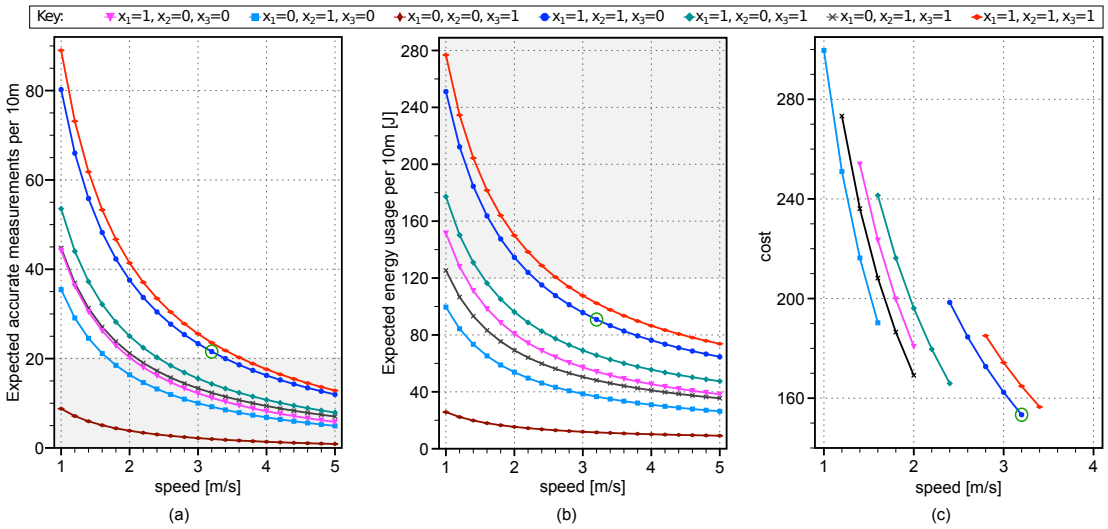


Figure 6.12: Verification results for requirement (a) R1, (b) R2, and (c) cost of the feasible configurations. The circled configuration is selected to adapt the system.

ical property *CP_R1* is the result of the refinement of R1 in a form appropriate for runtime verification. The solution placeholders *CP_R1Res* and *CP_R4Res* remain uninstantiated and should constantly be updated by the ENTRUST controller, once the verification result becomes available and the policy is satisfied, respectively.

6.1.5 Stage 5: Running the Self-Adaptive System

During this ENTRUST stage, the deployed self-adaptive system is dynamically adjusting its configuration in response to the observed internal and environmental changes. To this end, the controller executes a MAPE loop (cf. Figure 2.4) through which it: i) *monitors* the system and its environment through *sensors* and produces concrete system and environment stochastic models; ii) *analyses* these models using the probabilistic verification engine to check the compliance of the system with its requirements; iii) *plans* reconfiguration steps if this compliance is violated; and iv) *executes* this plan to achieve the new configuration of the managed system through *effectors*.

The use of continual verification within the ENTRUST control loop produces assurance evidence that underpins the dynamic synthesis of assurance arguments in the next stage of ENTRUST.

Example 6.6. Suppose that the UUV system from our running example comprises $n = 3$ sensors with: measurement rates $r_1 = 5, r_2 = 4, r_3 = 4$; energy consumed per

measurement $e_1 = 3, e_2 = 2.4, e_3 = 2.1$; and energy used for switching a sensor on and off $e_1^{\text{on}} = 10, e_2^{\text{on}} = 8, e_3^{on} = 5$ and $e_1^{\text{off}} = 2, e_2^{\text{off}} = 1.5, e_3^{\text{off}} = 1$, respectively.

Also, suppose that the current UUV configuration is $(x_1, x_2, x_3, sp) = (0, 1, 1, 2.8)$, and that sensor 3 experiences a degradation such that $r_3^{\text{new}} = 1$. The ENTRUST controller gets this new measurement rate through the *monitor* model. As the sensor rates differ from those in the knowledge repository, the guard *analysisRequired()* returns true and the **startAnalysis!** signal is sent. The *analyser* model receives this signal and invokes the *probabilistic verification engine*, whose analysis results for requirements R1–R3 are depicted in Figure 6.12. The analyse() action filters the results as follows: configurations that violate requirements R1 or R2, i.e., those from the shaded areas from Figure 6.12a and Figure 6.12b, respectively, are discarded. The remaining configurations are feasible, so their cost (R3) is computed for $w_1 = 1$ and $w_2 = 200$ (i.e., UUV speed is given higher priority that energy consumption of the sensors). The configuration minimising the cost (i.e., $(x_1, x_2, x_3, sp) = (1, 1, 0, 3.2)$ – circled in Figures 6.12a-c) is selected as the best configuration. Since the best and the current configurations differ, the analyzer invokes the planner to assemble a stepwise reconfiguration plan with which i) sensor 2 is switched on; ii) next, sensor 3 is switched off; and iii) finally the speed is adjusted to 3.2m/s. Once the plan is assembled, the executor is enforcing this plan to the UUV system. The adaptation results from Figure 6.12 provide the evidence required to update the assurance argument as described in the following section.

6.1.6 Stage 6: Synthesis of Dynamic Assurance Argument

In the final stage of ENTRUST, the assurance argument evolves as a result of the reconfiguration occurring in the self-adaptive system. When the system needs to adapt in response to a requirement violation or a significant change in the system or its environment, the argument is *backtracked* to *ReqsSatisfied*. Since ENTRUST self-adaptive systems can reconfigure themselves (justified by the goal *Reconfig*), a new goal *Reqs-Configuration* is instantiated as follows: either the controller selects a configuration that satisfies the QoS requirements associated with the normal system operation, or the controller selects a configuration that meets the failsafe requirement. In either case, the adaptation evidence produced in Stage 5 is used to fill in the corresponding place-

holders from the partially-instantiated assurance argument assembled in Stage 4. In the end, a fully-fledged GSN assurance argument is synthesised. For each new system configuration and adaptation evidence, an updated version of the assurance argument is assembled. This information is stored in the Knowledge repository (Figure 6.1) so that decision makers and auditors can understand and assess the current and past versions of the assurance argument.

Example 6.7. Consider again the partially-instantiated assurance argument pattern for our UUV system shown in Figure 6.11. In Example 6.6 we have seen that the currently used UUV configuration $(x_1, x_2, x_3, sp) = (0, 1, 1, 2.8)$ was unable to satisfy the system requirements due to changes in the measurement rate of its sensor $r_3^{\text{new}} = 1$. At that point, the critical property *CP_R1* is violated (i.e. *CP_R1Ver* is falsified) and the argument is *backtracked* to *ReqsSatisfied*, requesting the creation of a new version of the assurance argument. Since the controller found a new configuration $(x_1, x_2, x_3, sp) = (1, 1, 0, 3.2)$ that satisfies the requirements R1–R3, the goal *ReqsConfiguration* is instantiated with evidence produced from the quantitative analysis of the new configuration. Figure 6.13 depicts the complete assurance argument corresponding to this scenario.

6.2 Implementation

We adopted the ENTRUST methodology (Figure 6.2) and used several components for the development of an ENTRUST self-adaptive system (Figure 6.1). The trusted virtual machine is based on *ActivForms* [128], an approach for creating executable formal models, developed by our collaborators Prof. Danny Weyns and Usman Iftikhar. The probabilistic verification engine employs the verification libraries of the probabilistic model checker PRISM [149]. The UPPAAL model checker [18] is used for the development and verification of controller models in Stages 1 and 2. The Goal Structuring Notation [105] is used for the partial instantiation of the assurance argument pattern and its dynamic synthesis in Stages 4 and 6. We developed the components required for realising Stages 3 and 5 in Java, i.e., the *Sensors* and *Effectors*, and the interface for storing the assurance evidence and the updated assurance argument into the *Knowledge repository*. The open-source code and the full evaluation results presented in the next section are available at `http://www-users.cs.york.ac.uk/~simos/ENTRUST`.

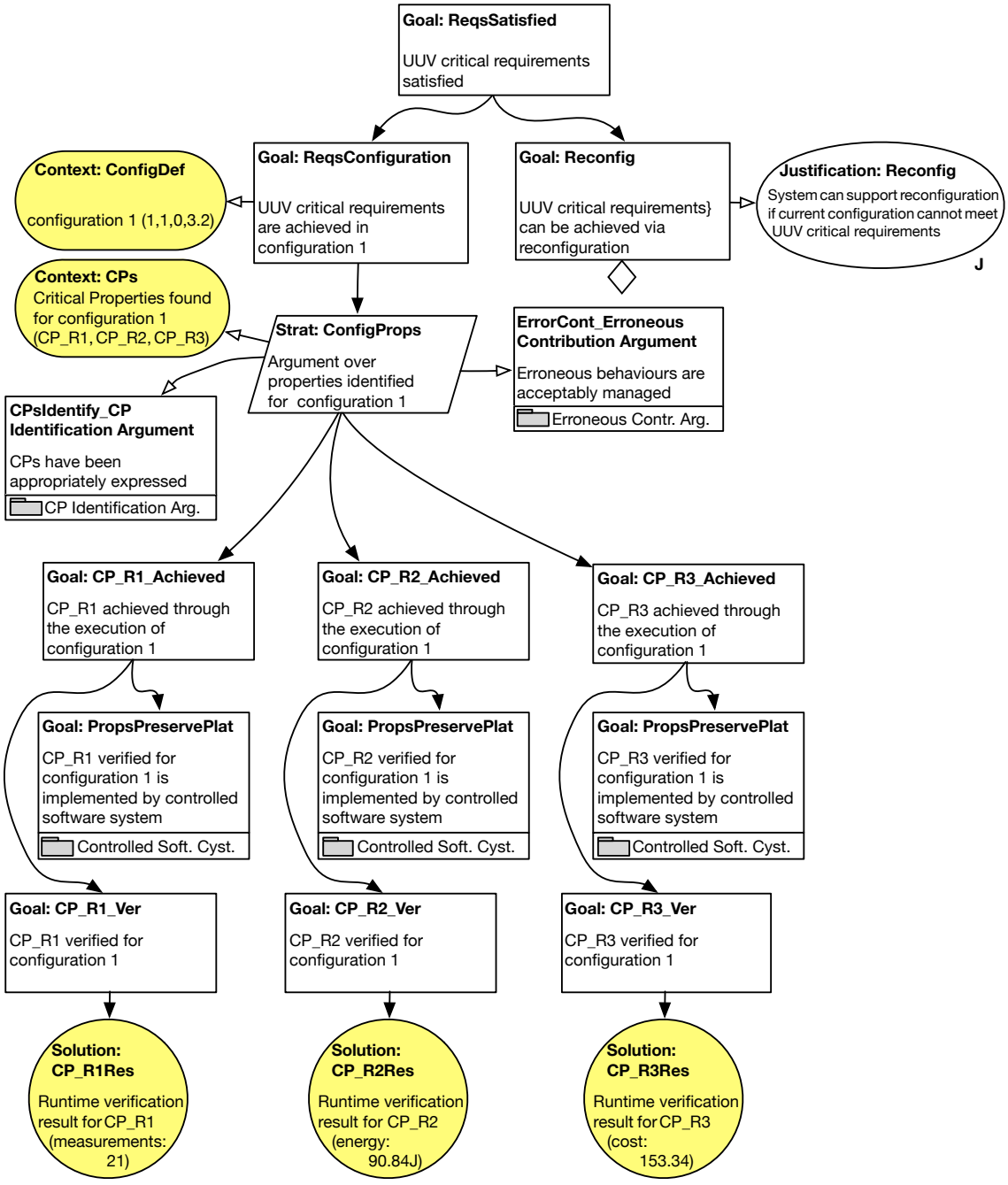


Figure 6.13: Fully-instantiated assurance argument for the UUV system. The shaded GSN elements correspond to the new configuration $(1, 1, 0, 3.2)$ selected by the ENTRUST controller (context *ConfigDef*); the link between this configuration and critical properties *CP_R1*, *CP_R2* and *CP_R3* (context *CPs*); and the verification evidence associated with this configuration (solutions *CP_R1Res*, *CP_R2Res* and *CP_R3Res*).

6.3 Evaluation

6.3.1 Research Questions

The aim of our experimental evaluation was to answer the following research questions:

RQ1 (Generality): Does ENTRUST support the development of high-integrity self-adaptive systems and dynamic assurance cases across application domains? Since we deal with self-adaptive systems from the safety-critical and mission-critical domains (e.g., UUV), ENTRUST is expected to support the development of systems from these domains. To establish the generality of ENTRUST we used it for the engineering of two self-adaptive software systems from different application domains.

RQ2 (Correctness): Are ENTRUST self-adaptive systems making the right adaptation decisions and generating valid assurance arguments? With this research question we examine whether controllers produced as part of the ENTRUST methodology reconfigure correctly the managed system. We also want to establish whether the adaptation evidence generated due to this reconfiguration is sufficient for synthesising a complete assurance argument.

RQ3 (Efficiency): Does ENTRUST provide design-time and runtime assurance evidence with acceptable overheads for realistic system sizes? We used this research question to investigate whether ENTRUST can scale with self-adaptive systems of varying sizes and complexity. To this end, we examined the overheads associated with various ENTRUST stages for these system sizes.

6.3.2 Experimental Setup

To evaluate ENTRUST and answer research questions **RQ1**–**RQ3**, we used the ENTRUST methodology (Figure 6.2) to engineer two prototype self-adaptive systems: i) the embedded UUV system used as a running example earlier in this chapter and ii) a foreign exchange (FX) service-based system with the description from Section 4.1 and the QoS requirements from Table 6.3. To assess ENTRUST for systems of increasing complexity, we applied it to UUV and FX system variants with the characteristics shown in Table 6.4. For the UUV system we varied the number of sensors. For the FX system we varied the number of implementations per service; we also assume that that only one third-party implementation is used at a time for each service.

Table 6.3: QoS requirements for the prototype FX self-adaptive system developed using ENTRUST

ID	Informal description
R1	"Workflow executions must complete successfully with probability at least 0.9."
R2	"The total service response time per workflow execution must not exceed 5s."
R3	"If requirements R1–R2 are satisfied by multiple configurations, the FX should use one of these configurations that minimises the function $cost = w_1C + w_2T$", (C and T are the cost and response time per workflow execution, respectively.)
R4	**(failsafe)** "If a suitable configuration is not identified within four seconds after a change in third-party service implementations is detected, the *Order* service invocation is disabled, so that the FX system does not carry out any trade using obsolete data (e.g., old exchange rates)".

For each case study, we initially developed the managed software system (without any self-adaptive capabilities). For the UUV system, as in our previous work (Chapters 3 and 4), we implemented a fully-fledged simulator using the open-source MOOS-IvP middleware [20], a widely used C++ platform for the implementation of autonomous applications on unmanned marine vehicles. For the FX system, we developed simple SOAP-based web services[5] and deployed these service on an Apache Tomcat web server[6]. Next, we implemented in Java the service-based system with the architecture shown in Figure 4.1. Afterwards, we applied the ENTRUST methodology to implement a controller and a partially-instantiated assurance argument pattern for each of the two considered managed systems (Stages 1–4). We then assembled the self-adaptive system by integrating the controller with the managed system.

We deployed each ENTRUST self-adaptive system in a realistic environment seeded with simulated changes specific to each domain. For the UUV system, changes included unexpected minor (up to 2%) and major (up to 10%) degradation in sensor measurement rates, complete sensor failures and recoveries from these problems. For the FX system, we considered small (up to 2%) and significant (up to 50%) patterns of increased response time and/or decreased reliability of third-party service implementations.

We analysed the adaptation decisions and the assurance arguments produced by ENTRUST in response to each of these unexpected events (Stages 5–6). All the experiments were carried out using a standard 2.6GhZ Intel Core i5 Macbook Pro computer with 16GB of memory and running Mac OSX 10.9 64-bit.

[5] `https://www.w3.org/TR/soap`
[6] `http://tomcat.apache.org`

Table 6.4: Characteristics of analysed UUV and FX system variants

Type	Details	
UUV speed	$sp \in [1m/s, 5m/s]$	
UUV Sensors	$n \in \{3, 4, 5, 6\}$	
Sensor rate	$r_i \in [1Hz, 10Hz]$	$1 \leq i \leq n$
FX alternative implementations per service	$n_i \in \{2, 3, 4, 5\}$	
Service implementation reliability	$r_{ij} \in [0, 1]$	$1 \leq j \leq n_i$
Service implementation response time	$t_{ij} \in [0, 10s]$	$1 \leq j \leq n_i$

6.3.3 Results and Discussion

RQ1 (Generality). To answer the first research question, we applied the ENTRUST methodology for the development of the UUV embedded system and the FX service-based system. The application of the ENTRUST methodology to the engineering of the UUV self-adaptive system is described in Sections 6.1.1– 6.1.6. Therefore, the remainder of this section focuses on the application of our methodology to developing the FX service-based system.

Development of the FX Service-Based System Using ENTRUST

Stage 1: Development of Verifiable Models

For developing the controller models, we specialised the ENTRUST event triggered MAPE model templates (Figure 6.3) and developed the models shown in Figure 6.14. The signal **newServiceCharacteristics?** is received by the monitor automaton from a sensor responsible for detecting changes in reliability and response time of third-party service implementations currently used by the managed system. When such change is significant, the guard *analysisRequired()* returns true and the analyzer automaton sends a **verify!** signal to the probabilistic verification engine. The engine carries out quantitative analysis and establishes which FX configurations meet requirements R1 and R2 and with what cost (cf. requirement R3). The analyzer automaton then filters the verification results and selects a configuration that satisfies R1 and R2 with minimum cost. If no such configuration can be found or the time threshold is exceeded, the Order service is disabled (cf. requirement R4). If the chosen configuration is not the same as the currently used configuration, the guard *adaptationRequired()* holds and the **startPlanning!** signal is sent to the planner automaton. The planner synthesises a

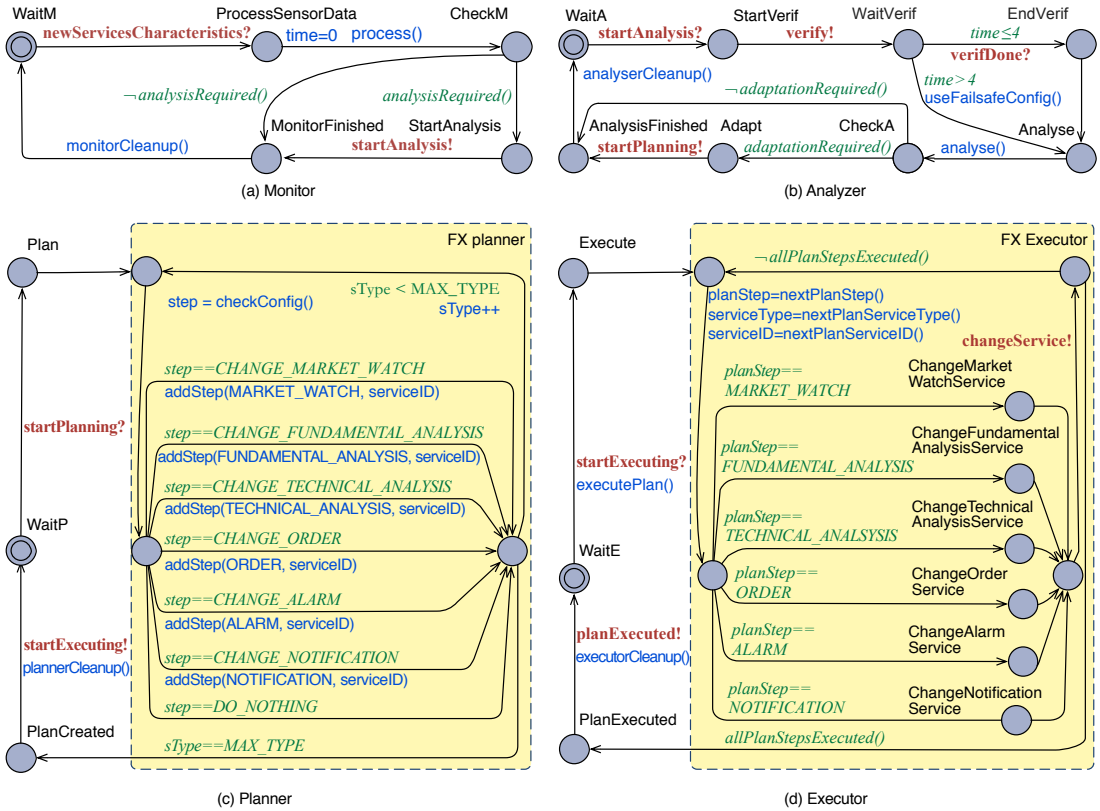


Figure 6.14: Instantiation of FX MAPE automata based on the event-triggered EN-TRUST model templates.

stepwise reconfiguration plan for switching to the new configuration by first activating the new third-party implementations for each service and then deactivating those that are no longer needed (as soon as the last request sent to them has been handled). This stepwise plan is implemented by the executor automaton, which communicates with the effector through the **changeService!** signal.

We model the behaviour of the FX system as a discrete-time Markov chain. The QoS requirements are formalised in PCTL as follows:

R1: $P_{\geq 0.9}[F \text{“success”}]$

R2: $R^{\text{“time”}}_{\leq 5}[F \text{“done”}]$

R3: minimise $w_1 C + w_2 T$

where $C = R^{\text{“cost”}}_{=?}[F \text{“done”}]$ and $T = R^{\text{“time”}}_{=?}[F \text{“done”}]$.

Since the failsafe requirement **R4** encodes a condition-action policy [127], it is not formalised in PCTL.

Stage 2: Verification of Controller Models

We used the UPPAAL model checker [18] to verify that the FX MAPE automata (Figure 6.14), satisfy all the correctness properties from Table 6.2. Following the recommended practice for this ENTRUST stage, we defined simple sensor, verification engine and effector automata (Figure 6.15). The sensor automaton has only one transition through which it sends the signal **newServicesCharacteristics!**, indicating changes in the behaviour of some service implementations. The effector automaton has also a single state and its transitions return to that state for each of the received signals **changeService?** and **planExecuted?**. The purpose of the verification engine automaton is to simulate: i) the quantitative analysis carried out when a change in the system is identified; and ii) the communication with the analyzer automaton through the signals **verify?** and **verifDone!**. As for the UUV, we specified a finite collection of appropriately instrumented sensor-verification engine automata in order to check all possible paths of the MAPE automata. Thus, we exercised all possible outcomes of the guards by simulating distinct and multiple changes to all services of the FX system.

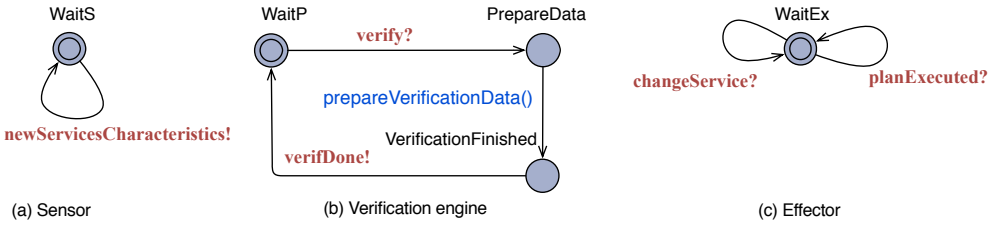


Figure 6.15: Auxiliary sensor, verification engine and effector automata used for verifying the generic controller properties from Table 6.2 for the FX system.

Stage 3: Controller Enactment

We instantiated the ENTRUST controller for the FX system by implementing the functionality of the abstract *Sensor*, *Effector* and *VerificationEngine* classes from the ENTRUST distribution. The sensor and effector classes synchronise with the *monitor* and *analyzer* automata through the signals **newServicesCharacteristics!**, and **changeService?** and **planExecuted?**, respectively. The communication with the managed system is made through using the relevant API methods of our FX simulator. The specialised effector class is shown in Listing 6.3. After receiving the **verify?** signal from the analyzer, the verification engine instantiates the parametric DTMC model and verifies the PCTL formulae associated with requirements R1 – R3. Once the engine completes its execution, it notifies the analyzer automaton through the **verifDone!** signal.

Listing 6.3: Effector class for the FX system.

```
1  package controller;
2
3  import java.util.HashMap;
4  import activforms.engine.ActivFORMSEngine;
5  import activforms.engine.Synchronizer;
6
7  public class Effector extends Synchronizer{
8
9    private ActivFORMSEngine vm;                                    // trusted VM
10   private int changeService, planExecuted;                        // signal(s)
11   private PrintWriter comm;                                       // comm handle
12
13
14   /** Constructor: create a new effector */
15   public Effector(ActivFORMSEngine vm, Object comm){
16     //assign handlers
17     this.vm           = vm;
18     this.comm         = (PrintWriter)comm;
19
20     //get signal(s) ID
21     changeService     = vm.getChannel("changeService");
22     planExecuted      = vm.getChannel("planExecuted");
23
24     //register signals
25     vm.register(changeService, this, "serviceType", "serviceID",
26                 "newConfig");
27     vm.register(planExecuted,  this, "newConfig");
28   }
29
30   /** Executed when receiving one of the registered signals....*/
31   @Override
32   public void receive(int channelID, HashMap<String, Object> data){
33     if (channelID == changeService){
34       Object serviceID   = data.get("serviceID");
35       Object serviceType = data.get("serviceType");
36       comm.println(serviceID, serviceType);
37       comm.flash();
38     }
39     else if (channelID == planExecuted){
40       //TODO: cleanup effector, if needed
41     }
42   }
43 }
44
```

Stage 4: Partial Instantiation of Assurance Argument Pattern

We developed a partial instance of the assurance argument pattern (Figure 6.8) for the self-adaptive FX system. For simplicity, the partially-instantiated assurance argument pattern in Figure 6.16 shows the details for requirement R2 and (failsafe) requirement R4. Similar reasoning applies for requirements R1 and R3. The outcome of formalising R2 into a form suitable for quantitative analysis is critical property *CP_R2*. The solutions *CP_R2Res* and *CP_R4Res* are uninstantiated and form placeholders that will be filled in with relevant evidence at runtime by the controller.

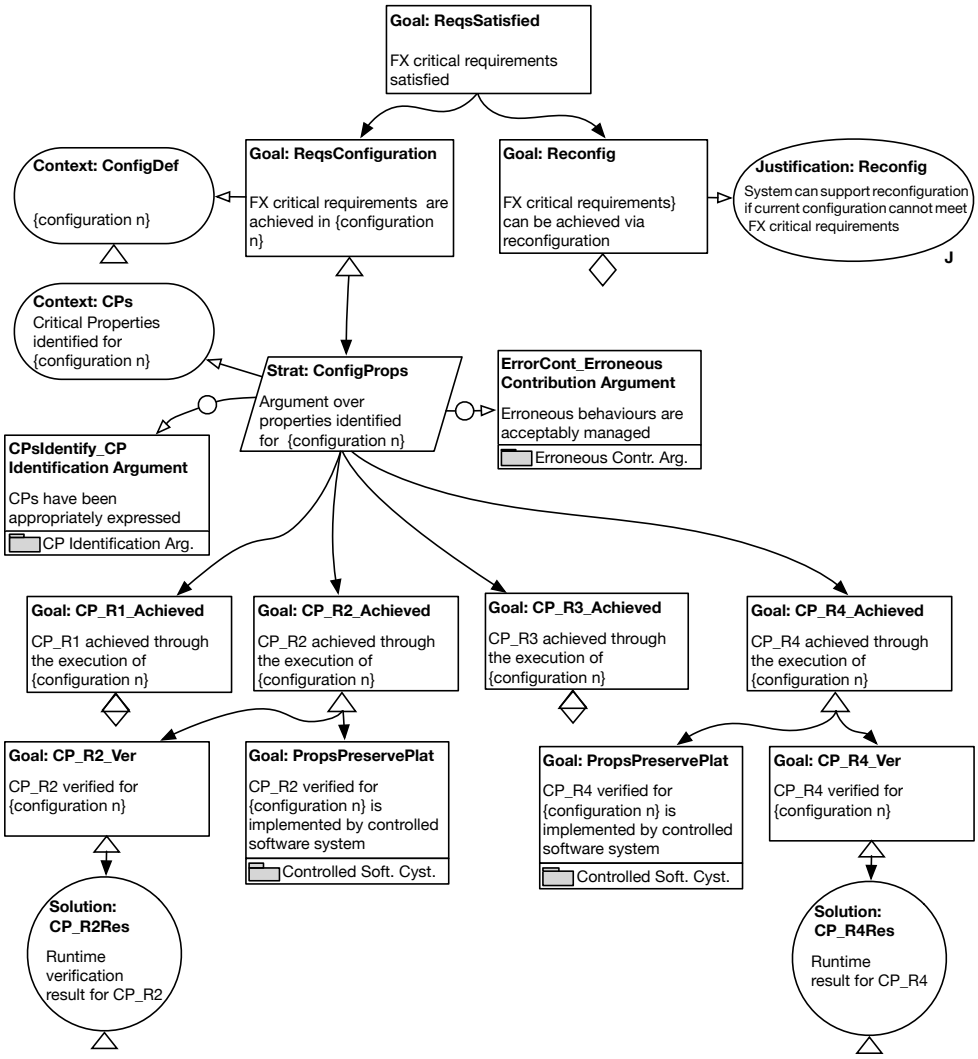


Figure 6.16: Partially-instantiated assurance argument for the FX system.

Stage 5: Running the Self-Adaptive System

We illustrate the adaptation decisions made by the ENTRUST controller using an unexpected event in which one of the selected service implementations experiences a significant degradation in its reliability. Suppose that there are three third-party implementations for each of the six FX services, and that these implementations have the characteristics shown in Table 6.5. Also, suppose that the current configuration of active service implementations is $(MW, TA, FA, AL, OR, NOT) = (MW1, TA3, FA3, AL3, OR1, NOT3)$ and $MW1$ experiences a significant service degradation.

The ENTRUST controller receives these updated services characteristics (reliability and response time) through the *monitor* model. Since the services characteristics are different from those in the *Knowledge repository*, the guard *analysisRequired()* holds and the **startAnalysis!** signal is sent. The *analyser* model, upon receiving the signal, invokes the *probabilistic verification engine*, whose results for requirements R1–R3 are depicted in Figure 6.17. Using the analyse() function, configurations that violate requirements R1 or R2, i.e., those from the shaded areas from the figure, are discarded. The remaining configurations are feasible, so their cost is computed for $w_1 = 1$ and $w_2 = 2$ (i.e., workflow response time has twice the priority of workflow cost). The configuration $(MW, TA, FA, AL, OR, NOT) = (MW2, TA1, FA3, AL3, OR3, NOT3)$ has the minimum cost and, thus, it is selected as the best system configuration; this con-

Table 6.5: Characteristics of the third-party service implementations of the FX system

ServiceID	Reliability	Time(s)	Cost(p)	ServiceID	Reliability	Time(s)	Cost(p)
MW1	0.976	5	0.5	OR1	0.995	25	0.6
MW2	0.995	10	0.5	OR2	0.95	20	1.3
MW3	0.996	10	1.5	OR3	0.95	10	1.4
TA1	0.998	6	0.6	AL1	0.915	15	0.6
TA2	0.990	18	1.3	AL2	0.990	9	0.9
TA3	0.985	14	1.0	AL3	0.990	6	1.2
FA1	0.998	23	1.6	NOT1	0.990	5	1.8
FA2	0.990	25	0.7	NOT2	0.990	8	0.5
FA3	0.990	8	1.2	NOT3	0.995	13	0.7

MW*:Market Watch, TA*: Technical Analysis, FA*: Fundamental Analysis

OR*: Order, AL*: Alarm, NOT*: Notification

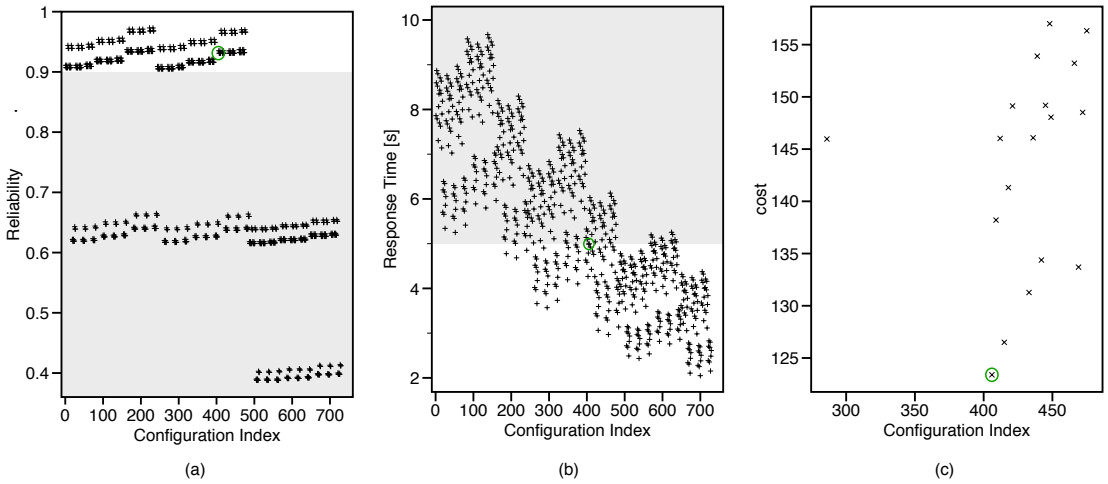


Figure 6.17: Verification results for requirement (a) R1, (b) R2, and (c) cost of the feasible configurations. The shaded areas in (a) and (b) denote configurations that violate R1 and R2, respectively. The circled configuration is selected to adapt the system.

figuration is circled in Figure 6.17. Given that the best and the current configurations are not the same, the *adaptationRequired()* guard holds and the *planner* is started by the *analyzer* using the **startPlanning!** signal. The planner then produces a stepwise reconfiguration plan with which i) $MW2$ replaces $MW1$; ii) $TA1$ replaces $TA3$; and iii) $OR1$ replaces $OR3$. When the plan is available, the *executor* receives the **startExecuting?** signal and is realising this plan to the FX system through sending the signal **changeService!** to effectors.

Stage 6: Synthesis of Dynamic Assurance Argument

Following the adaptation decision made in the previous ENTRUST stage, the controller must now use this assurance evidence to produce a new complete and valid assurance argument. Initially, the argument is backtracked to *ReqsSatisfied* due to the change observed in the service characteristics. The controller then re-instantiates the goal *ReqsConfiguration* using the evidence generated from the quantitative analysis of configuration $(MW, TA, FA, AL, OR, NOT) = (MW2, TA1, FA3, AL3, OR3, NOT3)$. The new assurance argument after the adaptation is shown in Figure 6.18.

We used ENTRUST to develop an embedded mission-critical system from the unmanned vehicle domain, and a service-based business-critical system from the exchange trade domain. Self-adaptation within these systems is underpinned by the verification of continuous- and discrete-time Markov chains, respectively. Although evaluation in additional domains is needed, these results suggest that ENTRUST can be used across application domains.

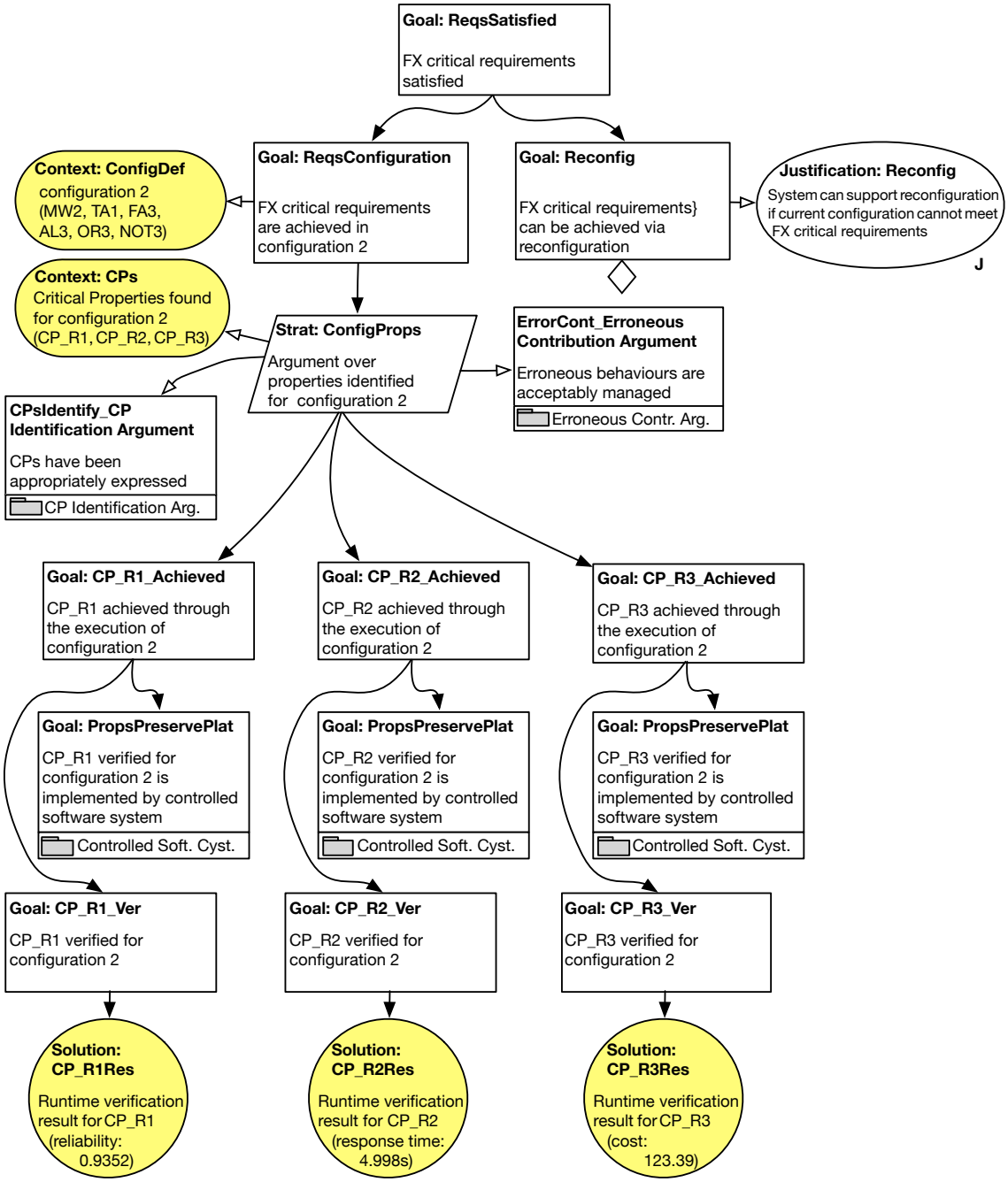


Figure 6.18: Fully-instantiated assurance argument for the FX system. The shaded GSN elements denote the new configuration $(MW2, TA1, FA3, AL3, OR3, NOT3)$ selected by the ENTRUST controller (context *ConfigDef*); the link between this configuration and critical properties *CP_R1*, *CP_R2* and *CP_R3* (context *CPs*); and the verification evidence associated with this configuration (solutions *CP_R1Res*, *CP_R2Res* and *CP_R3Res*).

RQ2 (Correctness). To answer this research question, we carried out a range of experiments that involved injecting into the prototype UUV and FX systems unexpected changes specific to each domain. For each system, we defined a concrete scenario involving seven changes each, shown in Table 6.6. For the UUV system we used the variant with $n = 3$ sensors, while for the FX system with used the variant with three third-party implementations per service. We assessed the adaptation decisions and the assurance arguments produced by the system in response to these events by performing two types of analysis.

For the former assessment, we established that the ENTRUST controller operated correctly. To this end, we verified that the sensors accurately identified and reported the changes leading to a correct monitor notification. Then, we confirmed that the new information was appropriately processed by the monitor which led the analyser to select a new configuration by carrying out the quantitative analysis using the probabilistic verification engine and filtering the results based on system QoS requirements. Finally, we validated that the planner assembled a correct plan for the new configuration and that this plan was implemented by the executor through effectors to the managed systems.

For the latter assessment, we determined the suitability of the ENTRUST assurance arguments. We started from the guidelines set by safety and assurance standards, which highlight the importance of demonstrating, using available evidence, that an assurance argument is *compelling*, *structured* and *valid* [200]. Also, we considered the fact that

Table 6.6: Changes in environment state of UUV system with 3 sensors and FX system with 3 third-party implementations per service

ID	UUV	FX
C1	Nominal	Nominal
C2	$r_3\downarrow$	$r_{11}\downarrow$, $r_{13}\downarrow$
C3	$r_3\leftrightarrow$	$r_{11}\leftrightarrow$, $r_{13}\leftrightarrow$
C4	$r_2\downarrow$	$r_{21}\downarrow$, $r_{22}\downarrow$
C5	$r_2\leftrightarrow$	$r_{21}\leftrightarrow$, $r_{22}\leftrightarrow$
C6	$r_2\downarrow$, $r_3\downarrow$	$t_{41}\uparrow$, $t_{42}\uparrow$
C7	$r_2\leftrightarrow$, $r_3\leftrightarrow$	$t_{41}\leftrightarrow$, $t_{42}\leftrightarrow$
Key	$\downarrow$: change (decrease) in environment characteristic	
	$\uparrow$: change (increase) in environment characteristic	
	$\leftrightarrow$: recovery of environment characteristic	

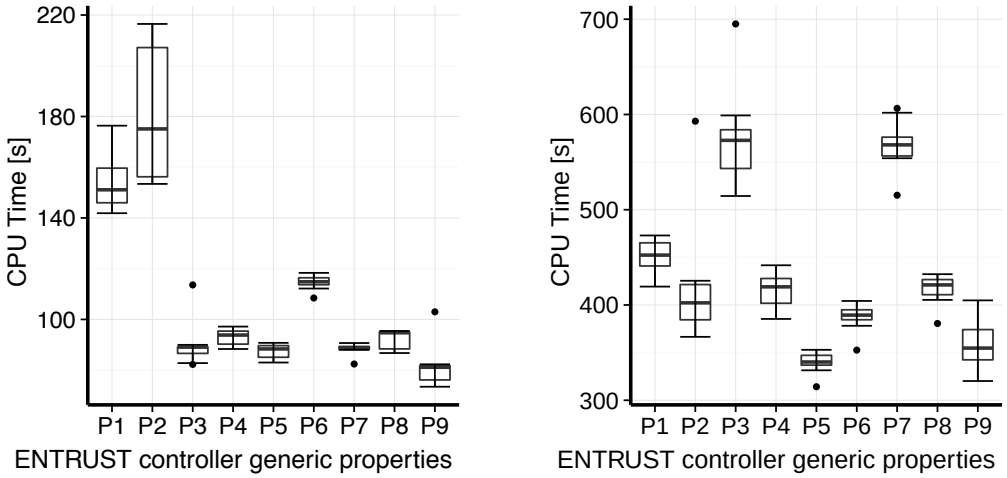
ENTRUST has been examined experimentally but has not been tested in real-world scenarios to produce the industrial evidence necessary before approaching the relevant regulator. However, our preliminary results show, based on formal design-time and runtime evidence, that the primary claim of ENTRUST assurance cases is supported by a direct and robust argument. Firstly, the argument assures the achievement of the critical requirements either based on a particular active configuration or through reconfiguration, while maintaining a failsafe mechanism. Secondly, the argument and patterns are well-structured and conform to the GSN standard [105]. Thirdly, ENTRUST provides rigorous assessments of validity not only at design time but also through-life, by means of monitoring and continuous verification that assess and challenge the validity of the assurance argument based on actual operational data. This continuous validity assessment is a core requirement for safety standards (e.g., for medical devices [181]).

In conclusion, subject to the limitations described above, our experiments provide strong empirical evidence that ENTRUST self-adaptive systems make the right adaptation decisions and generate valid assurance cases.

RQ3 (Efficiency). To assess the efficiency and scalability of ENTRUST we studied the overheads incurred by its design-time and runtime stages.

First, we analysed the CPU time taken to verify the controller properties from Table 6.2. in the second ENTRUST stage (Section 6.1.2). For each considered UUV and FX variant, we measured the time taken by the UPPAAL model checker [18] to verify these properties. We performed 10 independent measurements for each property. Figure 6.19 shows the time taken to verify these generic controller properties for a three-sensor UUV system, and for an FX system comprising two third-party implementations for each workflow service. For the UUV system, the CPU time consumed for most properties is below 2 minutes and the maximum time taken for verifying properties P1 and P2 does not exceed 4 minutes. The total time to verify all controller properties did not exceed 20 minutes. For the FX system, the verification of each property took between three and 12 minutes, and under 60 minutes in total. Since this is a design-time activity, the overheads for the verification of all controller properties are entirely acceptable.

Second, we examined the CPU time taken by the probabilistic verification engine to carry out the quantitative analysis during the ENTRUST self-adaptation stage (Section 6.1.5). For each system, and for change C2 from Table 6.6, we measured the time required for the probabilistic model checking of the QoS requirements for the possible system configurations. Figure 6.20 shows our results based on 10 independent runs. For

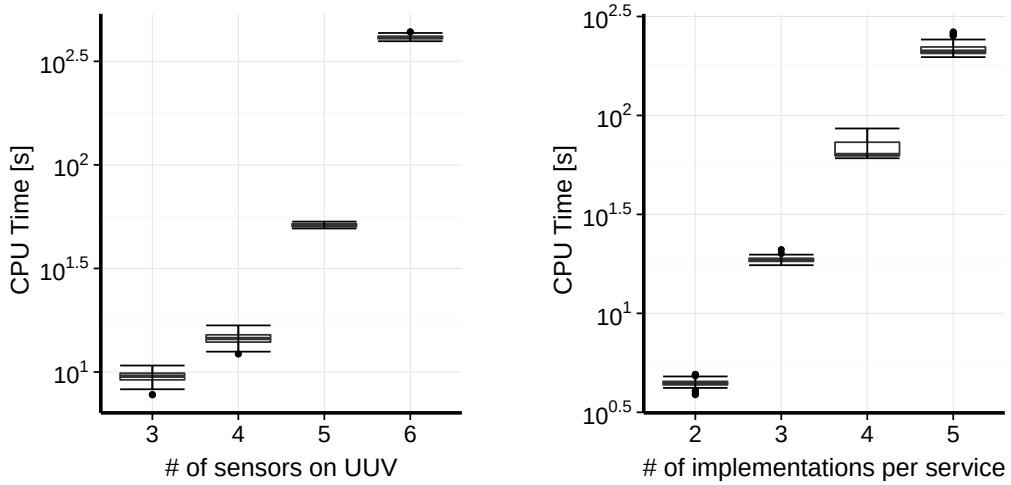


(a) UUV system (b) FX system

Figure 6.19: CPU time for the UPPAAL verification of the generic controller properties in Table 6.2 (box plots based on 10 independent runs)

the three-sensor UUV system and the FX system with two implementations per service, for instance, the CPU times have mean values below 10s and 3s, respectively. Although these runtime overheads limit the applicability of ENTRUST to systems in which reconfigurations are infrequent (i.e., changes are not in the order of seconds), many real-world systems meet this condition. Furthermore, these critical systems have failsafe configurations that they can temporarily realise during the infrequent reverifications of the ENTRUST stochastic models.

Finally, we ran experiments to assess the scalability of ENTRUST by measuring the increase in runtime overheads with the system size and with the number of alternative configurations. The UUV and FX variants used in these additional experiments comprised up to six sensors and up to five implementations per service, respectively. We used the nominal values for the environment state of these variants (i.e., change C1 from Table 6.6). Figure 6.20 shows the box plots for these system variants based again on 10 independent runs. We observed that the CPU time increases exponentially with these system characteristics, a typical problem for model checking known as state explosion [48]. This makes our current realisation of ENTRUST suitable for small to medium-sized systems. Recent advances in the area (Section 2.2.3) and our recent work presented in Chapters 3 – 5 can be integrated with ENTRUST to extend its applicability to larger system sizes.



(a) UUV system (b) FX system

Figure 6.20: CPU time for the runtime probabilistic model checking of the QoS requirements after changes (box plots based on 10 system runs)

6.3.4 Threats to Validity

Construct validity threats may be due to the assumptions made when implementing the prototype UUV and FX systems, and in the development of the stochastic models and requirements for these systems. To mitigate these threats, we implemented the two systems using the well-established UUV software platform MOOS-IvP and (for FX) standard Java web services deployed in Tomcat/Axis. The model and requirements for the UUV system are based on a validated case study that we are familiar with from previous work (Chapter 3), and those for the FX system were developed in close collaboration with a foreign exchange expert and also used in our previous work (Chapter 4).

Internal validity threats can originate from how the experiments were performed, and due to researcher subjectivity when interpreting the results. To address these threats, we reported results over multiple independent runs; we worked with a team comprising experts in all the key areas of ENTRUST(self-adaptation, formal verification and assurance cases); and we made all experimental results publicly available to enable replication on the project webpage `http://www-users.cs.york.ac.uk/~simos/ENTRUST`.

External validity threats may be due to the use of only two software systems in our evaluation. To reduce this threat, we selected these systems from different domains with different characteristics. The evaluation results provide evidence that ENTRUST supports the development of high-integrity self-adaptive solutions with assurance cases

for the two different settings. Nevertheless, additional evaluation is required to confirm generality for domains with characteristics that differ from those in our evaluation (e.g., different timing patterns and types of requirements and disturbances).

6.4 Related Work

The provision of evidence that self-adaptive software systems comply with strict QoS requirements is an area of active research [196, 208]. Nevertheless, assurance has become a vital element concerning the entire lifecycle of self-adaptive systems only recently [40, 55, 56, 205]. As such, most of the existing approaches are confined to providing correctness evidence for specific aspects of the self-adaptive system. ENTRUST is the first tool-supported and fully-fledged methodology, spanning across design-time and runtime, for the engineering of high-integrity self-adaptive software systems. In the following paragraphs we review existing illustrative approaches to providing evidence for self-adaptive software. We also explain how ENTRUST employs and extends previous work done by the ENTRUST team.

Assurance approaches. *Formal proof* is a technique used to develop theorems to establish key properties (e.g., safety, liveness, deadlock freeness) of a self-adaptive software system, including its controller and its processes after reconfiguration [64, 218]. For example, theorems have been used to assure the atomicity of business processes [212], and safety and liveness properties of self-adaptive systems [218]. Although formal proof is a possible type of evidence within the ENTRUST assurance argument, existing formal proof approaches require complete specifications of the system and its environment. Thus, they cannot be applied in systems in which these elements become known only at runtime.

Model checking [63, 64, 217] and *runtime quantitative verification* [38, 68, 98] have been used to produce assurance evidence for various aspects of the self-adaptation process. We provided a comprehensive review of techniques using RQV in Section 2.2.3. ENTRUST, as described in Section 6.1, builds upon these techniques to establish key correctness properties of the MAPE controller at design-time, and to obtain adaptation assurance evidence at runtime.

Control theoretical approaches have been used to develop controllers at runtime and provide control theoretical evidence for several system properties including stability and robustness [78, 140]. This is an interesting research direction and despite the promising early results, its applicability to real-world (or realistic) self-adaptive software systems

remains an open question. In contrast, ENTRUST employs established techniques for modelling and analysing software systems and assuring their required properties.

ENTRUST Foundation. ENTRUST builds on our and our collaborators' previous work on: formally verified control loops [99, 128], runtime quantitative verification [32, 38] and dynamic safety cases [59]. We integrated and enhanced this research with the necessary features to develop a fully-fledged methodology for engineering trustworthy self-adaptive systems. The novel contributions of ENTRUST include:

1) a formally verifiable controller architecture for self-adaptive software that integrates RQV [32, 38] into the controller modules [99, 128];

2) a set of generic correctness properties that ENTRUST controllers must satisfy;

3) a realisation of dynamic assurance arguments proposed in our preliminary work [59];

4) a methodology for the development of trustworthy self-adaptive software underpinned by formal assurance cases;

5) an extensive evaluation using two case studies from different application domains.

6.5 Summary

In this chapter we introduced ENTRUST, the first end-to-end tool-supported methodology for the engineering of trustworthy self-adaptive software systems. The ENTRUST methodology supports the entire lifecycle of a self-adaptive system, and includes methods for the development of verifiable controllers for self-adaptive systems, for the generation of design-time and runtime assurance evidence and for the runtime instantiation of an assurance argument pattern that we specifically developed for these systems. Engineers that adopt the ENTRUST methodology to implement self-adaptive systems must: (1) develop verifiable models for the controller of the system and parametric stochastic models that capture the uncertainty associated with the managed system and the environment in which it operates; (2) verify that the controller models operate as expected by establishing their compliance with a set of key correctness properties; (3) enact the controller by integrating the models developed earlier with ENTRUST reusable components (i.e., the trusted virtual machine and the probabilistic verification engine); (4) instantiate partially the assurance argument pattern using the evidence developed from stage 2, leaving placeholders for assurance goals that can only be resolved at runtime;

(5) deploy and run the self-adaptive system; and (6) enable the system to dynamically update the assurance argument using evidence produced during adaptation.

We evaluated ENTRUST using two self-adaptive systems, an embedded system for unmanned underwater vehicles and a foreign exchange service-based system. We assessed the generality, correctness, and efficiency of ENTRUST. Our results provide evidence that the ENTRUST methodology is suitable for the engineering of trustworthy self-adaptive system from different application domains. Through extensive experiments that involved deploying the two prototype self-adaptive systems in scenarios seeded with failure patterns specific to each domain, we established (1) the correct execution of the ENTRUST controller; and (2) the validity of the generated assurance arguments. Finally, we examined the overheads associated with the ENTRUST stages responsible for generating assurance evidence. We determined that the overheads for the design-time verification of controller properties and the runtime quantitative analysis of possible system configurations are acceptable for small-to-medium self-adaptive systems. For larger systems, our approach is affected by the state explosion problem [48]. However, recent advances in model checking and runtime quantitative verification (cf. Section 2.2.3) and the RQV variants we introduced earlier in the thesis reduce the overheads for this class of systems and enable its adoption in runtime settings.

Chapter 7

Conclusion and Future Work

This thesis addressed major limitations of runtime quantitative verification (RQV), a formal approach to implementing the closed-loop control of self-adaptive systems. RQV uses quantitative model checking of stochastic models describing the behaviour of a self-adaptive system and its environment, to rigorously identify or predict violations of QoS requirements, and to drive adaptation towards restoring or maintaining compliance with these requirements, respectively.

Despite its strengths, the approach is affected by state explosion, a common model checking problem where the size of the model increases exponentially with the size of the modelled system. This makes the computation and memory overheads of RQV unacceptable for large self-adaptive systems. Even when RQV can analyse fast enough the model associated with a specific system configuration, the extremely large configuration spaces of many typical self-adaptive systems render the analysis of all alternative system configurations infeasible. Considering these challenges, we defined the following research hypothesis:

> *Given the representation of key aspects of a self-adaptive system as Markov models and a set of QoS requirements defined in suitable probabilistic temporal logics, efficient runtime quantitative verification techniques can provide guarantees that the system continues to satisfy its QoS requirements in the presence of changes, for much larger systems and with much lower overheads than the standard RQV approach.*

The new RQV variants and methodology that we devised as part of this thesis support the research hypothesis and reduce the RQV overheads by several orders of magnitude. Our contributions improve the efficiency and scalability of RQV and extend its applicability to large, more complex and distributed software systems. We have shown that adapting methods from other software engineering areas (Chapter 3) and employing evolutionary algorithms (Chapter 4) can reduce the RQV overheads significantly. Also, we proposed a decentralised RQV variant that makes the technique suitable for distributed self-adaptive systems (Chapter 5). Finally, we defined an RQV-based methodology for engineering trustworthy self-adaptive systems (Chapter 6).

Notwithstanding the RQV advances achieved by this thesis (Chapters 3–6) and recent research (Section 2.2.3), our findings showed that the suitability of an RQV variant depends on the characteristics of each self-adaptive software system (e.g., centralised or distributed, size of configuration space, available resources). Thus, none RQV variant is a "silver bullet", i.e., no single RQV variant can handle all types of self-adaptive systems. There are still ample opportunities for research to improve the efficiency of RQV and to extend its applicability to other types of self-adaptive systems (e.g., self-interested distributed systems) and stochastic models (e.g., Markov decision processes, interval Markov chains). We envisage the development of a collection of RQV-based techniques which can be combined or used interchangeably depending on the self-adaptive system's characteristics, the type of stochastic model, the deployment platform, and the reconfiguration needs. Finally, the RQV-based methodology for engineering trustworthy self-adaptive systems can be extended with other approaches for the provision of assurances (e.g., formal proofs, model checking, testing). We anticipate that the methodology can form the basis of a standardised framework for the engineering of trustworthy self-adaptive systems.

In the following sections, we summarise our contributions, explain how each contribution supports the research hypothesis, and highlight directions for future research.

7.1 Efficient RQV Using Software Engineering Methods

7.1.1 Research Contributions

We integrated RQV with adapted versions of three efficiency improvement techniques previously used in other areas of software engineering (cf. Section 3.1). Firstly, we integrated RQV with caching which involves the storage of recent RQV analysis results and their reuse if the same environmental changes occur again in the future. Secondly,

RQV enhanced with limited lookahead uses spare CPU cycles between adaptation steps to pre-verify states of the system that are likely to arise in the future. When a change occurs, if the current system state has already been verified, the verification results are simply retrieved and no further RQV analysis takes place. Finally, nearly-optimal reconfiguration stops early the search for the next configuration as soon as a valid configuration has been found and a "near optimality" stopping criterion is satisfied.

The evaluation of these three RQV variants described in Section 3.3 indicates that all the variants improve the response time of RQV, but each has some trade-offs. Caching and limited lookahead consume extra memory for storing the recent verification results. The latter technique also needs additional CPU for the pre-verification process. Nearly-optimal reconfiguration, however, is very lightweight (compared to standard RQV), but its stopping criterion might cause the selection of sub-optimal configurations.

7.1.2 Further Research Directions

Two further research directions are worth exploring in order to improve the effectiveness of the RQV variants described above:

- Use of more sophisticated variants of the proposed techniques. For caching this includes using other cache replacement policies (e.g., adaptive replacement cache [172]). Alternative functions for the identification of "nearby" states (e.g., Manhattan distance) could be used for limited lookahead. For nearly-optimal reconfiguration it is worth investigating whether configurations that are closer to the optimal configuration could be obtained by using adaptive lenience policies which take into consideration the state of the environment, or the time and/or resources available for completing a reconfiguration.

- The RQV variants described in Section 3.1 could be integrated with compositional and incremental quantitative verification (Sections 2.2.3.1 and 2.2.3.2), so as to improve the RQV efficiency beyond what each of these classes of techniques can achieve on its own. Furthermore, these RQV variants could be combined into a collection of adaptive techniques, which could be used interchangeably or depending on the reconfiguration needs and the available resources.

7.2 Improving RQV Efficiency Using Evolutionary Algorithms

7.2.1 Research Contributions

We introduced EvoChecker, a search-based approach that improves the efficiency of RQV and extends the range of systems that RQV can handle by incorporating evolutionary algorithms into the verification process. More specifically, EvoChecker uses a probabilistic model template to encode the configuration parameters of a self-adaptive system (cf. Section 4.1.1), specifies QoS requirements as constraints and optimisation objectives, and searches the configuration space to find effective system configurations. We developed a human-in-the-loop EvoChecker (cf. Section 4.1.3) that is capable of identifying Pareto optimal configurations and adapts a system after the new configurations are validated by system experts. We also devised an automated EvoChecker (cf. Section 4.1.4) that can find configurations which optimise a predetermined trade-off between the objectives, employs an external archive to store recent verification results that are used to seed the search for a new configuration and updates the archive using suitable strategies.

We evaluated each EvoChecker variant using two case studies from different application domains (cf. Section 4.3). Our findings demonstrate the effectiveness, applicability and flexibility of each variant. The human-in-the-loop EvoChecker identifies Pareto optimal configurations and helps system experts to make informed adaptation decisions. In the automated EvoChecker, we found that combining the external archive with a suitable updating strategy helps this EvoChecker variant to achieve significant reductions in RQV overheads and to find effective system configurations very fast.

7.2.2 Further Research Directions

Several research directions are worth exploring:

- The applicability of EvoChecker could be extended to other modelling formalisms and verification logics by exploiting other established quantitative model checkers such as UPPAAL [18] and MRMC [133].

- It would be interesting to enhance EvoChecker with other evolutionary and natured-inspired multi-objective optimisation algorithms like evolutionary strategies, particle swarm optimisation and ant-colony optimisation.

- Evaluating further the performance of EvoChecker using a broader range of system changes, and assessing its applicability in other domains (e.g., robotics, cloud computing) is also an interesting research direction.

- The performance of the human-in-the-loop EvoChecker could be improved using an archive and suitable updating strategies (e.g., clustering and classification). We highlighted the potential of this idea and the limited research in this direction in Section 4.1.4.

- The integration of the EvoChecker approach with recent RQV advances (Section 2.2.3) and/or our approaches from Chapter 3 could improve further its performance.

7.3 Extending RQV With Decentralised Control Loops

7.3.1 Research Contributions

We introduced DECIDE, an RQV-driven approach that decentralises the control loops in distributed self-adaptive software systems (cf. Section 5.1). Each component within a DECIDE-based system uses RQV locally to analyse its capabilities and establish a summary of contributions it can make to help satisfying system-level QoS requirements. This QoS capability summary is shared with peer components. Next, the component carries out a decentralised selection of component contribution-level agreements (CLAs) which establishes the compliance with system-level QoS requirements. Finally, the component uses an RQV-driven local control loop to assure that it meets its CLA and local QoS requirements. Infrequently, the component is unable to achieve its CLA due to major changes. These events trigger a new local analysis, sharing of contributions summary with peers, and selection of new local CLAs.

We validated DECIDE using a simulated embedded system from the unmanned underwater vehicle domain and demonstrated its efficiency, effectiveness and scalability (cf. Section 5.3). Compared to centralised RQV-driven control loops, DECIDE is able to reconfigure a distributed self-adaptive system with overheads that are several orders of magnitude lower. DECIDE can also scale with insignificant increase in overheads to systems of much larger sizes (up to 32 UUVs in our case study), without introducing a single point of failure, and with reasonable increase in system-level costs (18-21% in our case study).

7.3.2 Further Research Directions

Several research directions deserve further exploration:

- DECIDE could be extended to support interface models as component QoS attributes, using assume-guarantee verification to verify system QoS properties [130].
- It would be interesting to assess the effectiveness and scalability of DECIDE in other types of distributed systems, including service-based and cloud-deployed software systems with heterogeneous components, and systems with larger component models (e.g., UUV systems with more sensors per UUV).
- Exploring the extension of DECIDE to competitive distributed self-adaptive systems is another dimension worth exploring. Systems from this category share common system-level requirements but at the same time have conflicting objectives. This aspect introduces additional complexity for the establishment of component-level agreements.
- Given that DECIDE is a generic approach, the development of a DECIDE library and an associated application programming interface will help the DECIDE integration with other distributed self-adaptive systems with minimal development effort.
- DECIDE could be integrated with Kevoree [84] or DEECo [28] to provide a complete platform for the engineering of distributed self-adaptive systems.

7.4 Engineering Trustworthy Self-Adaptive Software Systems

7.4.1 Research Contributions

We devised ENTRUST, the first end-to-end methodology for the engineering of trustworthy self-adaptive software systems (in which adaptation decisions are driven by RQV) and the dynamic generation of their assurance cases (cf. Section 6.1). ENTRUST spans across the entire lifecycle of a self-adaptive system. At design-time, system experts develop parametric stochastic models of the managed system and its environment. Models of the controller of the self-adaptive system are also developed and

verified to establish their compliance with key correctness properties. Next, the controller is enacted by integrating the controller models with the parametric stochastic models and the reusable ENTRUST components (i.e., probabilistic verification engine and trusted MAPE virtual machine). Any assurance evidence available at design-time is used for the partial instantiation of an assurance argument pattern. This includes evidence generated from the verification of the controller models and correctness evidence for the ENTRUST reusable components. The self-adaptive system is then deployed and dynamically reconfigures itself when environmental or system changes occur. The assurance evidence produced due to system adaptation is used to fill in the placeholders from the partially instantiated assurance argument and to derive the complete assurance argument of the self-adaptive system.

We established the applicability of ENTRUST by applying it for the development of two self-adaptive systems, an embedded UUV system and a foreign exchange service-based system (cf. Section 6.3). We also carried out extensive experiments to evaluate the correctness and efficiency of ENTRUST. Our findings confirm the validity of the generated assurance arguments, and the capability of ENTRUST controllers to reconfigure self-adaptive systems and execute the correct adaptation decisions. Finally, we analysed the overheads associated with the design-time and runtime stages generating assurance evidence and confirmed that these overheads are acceptable for small-to-medium self-adaptive systems. In larger systems, ENTRUST can employ recent RQV efficiency improvement techniques (Section 2.2.3) or our contributions described in Chapters 3–5.

7.4.2 Further Research Directions

Future research directions include:

- It would be interesting to evaluate the applicability of ENTRUST to other systems and application domains that include different features than the systems in our evaluation (e.g., different timing patterns and types of requirements and disturbances).

- ENTRUST could be extended to support runtime verification of both QoS requirements and functional requirements.

- The runtime overheads of ENTRUST could be reduced by exploiting recent advances in RQV (Section 2.2.3) or our contributions presented in Chapters 3–5.

7.5 Prototype Self-Adaptive Software Systems

We developed two prototype self-adaptive software systems as part of this research project. The former, described in Section 2.2.1.1, is a simulator of unmanned underwater vehicles (UUVs) from the domain of embedded and robotic systems. We built the simulator using MOOS-IvP [20], an open-source middleware for the implementation of autonomous applications on unmanned marine vehicles. The latter system, introduced in Section 4.1, is a service-based system from the domain of foreign exchange (FX) trading. We developed this system in collaboration with a European foreign exchange brokerage company. For each system, we have provided a general description, specified the observable and configurable system characteristics of interest (e.g., behaviour or architecture) and listed its QoS requirements. Based on this information, we have built a stochastic model of the system (in an appropriate Markov model variant) and formalised its QoS requirements using a suitable probabilistic temporal logic variant. Finally, we have developed a prototype implementation and employed it for the evaluation of our contributions.

In the future, these self-adaptive systems could be evolved further by extending the range of their observable and configurable characteristics, and by specifying additional requirements for verification. It is also worth improving the usability of these prototypes (e.g., providing APIs) so that other techniques and approaches can be examined with minimal development effort. We envisage that the FX and UUV systems could be used as exemplars[1] by other researchers in the area of self-adaptive systems.

[1] `https://www.hpi.uni-potsdam.de/giese/public/selfadapt/`

Appendix A

Sequential Strategy Module for the MarketWatch FX Service

```
1  const int STEPMAX = 4;

2  // Sequential strategy for Service #1: Market Watch
3  module MarketWatchSeqStrategy
4    mw: [0..6] init 0;// MW state
5    step: [1..4] init 1;// step
6    // Start MW
7    [startMW] mw=0 -> 1.0: (mw'=1);
8    // Check services
9    [checkMW1] mw=1 & step=1 -> (ex1=1|ex1=2)?1:0 : mw=2 +
10                               (ex1=3|ex1=4)?1:0 : mw=3 +
11                               (ex1=5|ex1=6)?1:0 : mw=4;
12   [checkMW2] mw=1 & step=2 -> (ex1=3|ex1=5)?1:0 : mw=2 +
13                               (ex1=1|ex1=6)?1:0 : mw=3 +
14                               (ex1=2|ex1=4)?1:0 : mw=4;
15   [checkMW3] mw=1 & step=3 -> (ex1=4|ex1=6)?1:0 : mw=2 +
16                               (ex1=2|ex1=5)?1:0 : mw=3 +
17                               (ex1=1|ex1=3)?1:0 : mw=4;
18   [checkMW4] mw=1 & step>3 -> 1.0: (mw'=5);
19   // Run services
20   [runMW1] mw=2 -> x11 × r11 : (mw'=6) +
21                    x11=1?1-r11:1.0 (mw'=1) & step'=min(STEPMAX,step+1);
22   [runMW2] mw=3 -> x12 × r12 : (mw'=6) +
23                    x12=1?1-r12:1.0 (mw'=1) & step'=min(STEPMAX,step+1);
24   [runMW1] mw=3 -> x13 × r13 : (mw'=6) +
25                    x13=1?1-r13:1.0 (mw'=1) & step'=min(STEPMAX,step+1);
26   // End Market Watch service
27   [failedMW] mw=5 -> 1.0:(mw'=0);
28   [succMW] mw=6 -> 1.0:(mw'=0);
29 endmodule
```

Figure A.1: Sequential strategy module for the MarketWatch service used by the FX system.

Appendix B

Dynamic Power Management System

We use the software-controlled dynamic power management (DPM) system with the architecture shown in Figure B.1, adapted from [174, 188]. The system consists of a *service provider* that handles requests generated by a *service requester* and stored in two *request queues* of different priorities. The service provider has four states associated with different power usage, i.e., *busy, idle, standby* and *sleep*. Figure B.1 depicts the power usage of each state (in watts), the possible transitions between states, and the energy consumed by each transition (in joules). These values are taken from [174], and correspond to a Fujitsu disk drive.

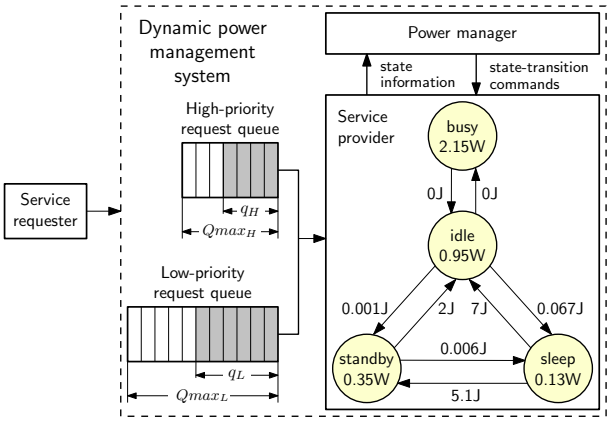


Figure B.1: Dynamic power management system

Table B.1: Average service-provider transition times

State transition	Average time (s)	State transition	Average time (s)
idle → *standby*	0.4	*standby* → *idle*	1.2
idle → *sleep*	0.67	*sleep* → *idle*	1.6
standby → *sleep*	0.3	*sleep* → *standby*	0.6

When the service provider is in the *busy* state, it processes requests as follows. If the high-priority queue contains $q_H > 0$ requests, then a high-priority request is processed. Otherwise, if the low-priority queue contains requests (i.e., if $q_L > 0$), a low-priority request is handled. After handling the last request (i.e., when both queues become empty), the service provider automatically transitions to the *idle* state. The transitions from *idle* to *busy* are also automatic, and occur whenever the empty-queue DPM system receives a new request. In contrast, all the other transitions are controlled by a software *power manager* that aims to reduce power use while maintaining an acceptable service level for the system. We use the real values from [174] for the state transition times (Table B.1) and the request service rate (i.e., 125s^{-1}). Figure B.2 shows an excerpt of the CTMC model of the DPM system specified in the PRISM modelling language.

The DPM system is required to adapt to changes in the arrival rates of the high-priority and low-priority requests so that the QoS requirements in Table B.2 are satisfied. To this end, DPM must select i) the capacity of the request queues, $Qmax_H$ and $Qmax_L$; ii) one of two alternative power managers; and iii) the parameters associated with the selected power manager.

Table B.2: QoS requirements for the DPM system

ID	Description
R1	The steady-state utilisation of the high-priority queue should be less than 90%
R2	The steady-state utilisation of the low-priority queue should be less than 90%
R3	The system should operate with minimum steady-state power utilisation
R4	The number of requests lost at the steady state should be minimised
R5	The capacity of both queues should be minimised

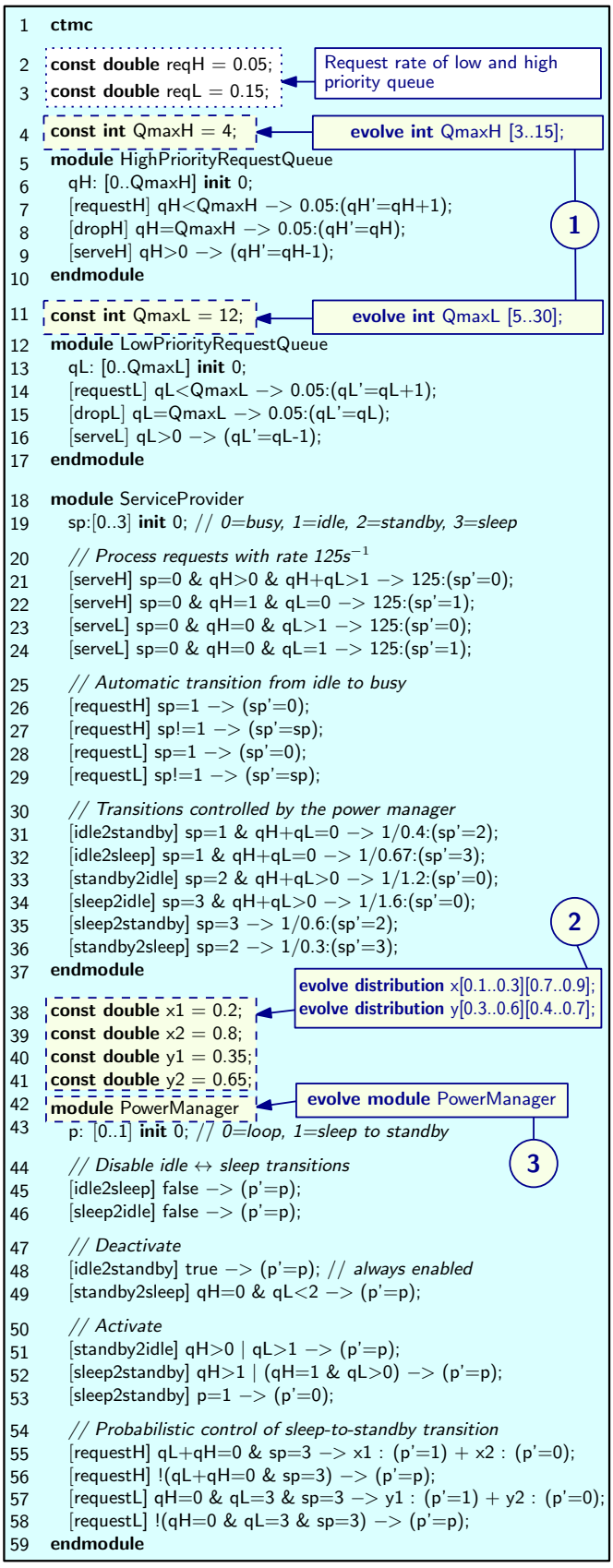


Figure B.2: CTMC model of the DPM system; ①–③ represent EvoChecker extensions of the PRISM modelling language

Glossary

Adaptive system	An open-loop system in which adaptation is manual and typically initiated by a human operator (as in the context of traditional software maintenance).
Self-adaptive system	A close-loop system capable of assessing the state of the environment and the system itself, and modifying its architecture or configuration when it detects a deviation from the expected behaviour, or when better functionality or performance is possible.
Managed software system	A component of a self-adaptive system that is responsible for the execution of the system's main functionality.
Controller	A component of a self-adaptive system that implements the MAPE-K control loop. The controller monitors the managed system and its environment and adapts the architecture or configuration of the managed system after environmental and internal changes.

References

[1] PRISM probabilistic model checker web site. http://www.prismmodelchecker.org/other-tools.php.

[2] Future Internet Assembly. Research Roadmap Towards Framework 8: Research Priorities for the Future Internet. http://fisa.future-internet.eu/images/0/00/FINAL_COMBINED_ROADMAP_VERSION_2.0.pdf, July 2012.

[3] Networked European Software and Service Initiative. Strategic Research and Innovation Agenda. http://www.nessi-europe.com/Files/Private/NESSI_SRIA_Final.pdf, April 2013.

[4] Aerospace, Aviation and Defence Knowledge Transfer Network. Autonomous systems: Opportunities and challenges for the UK. http://digital-library.theiet.org/content/conferences/10.1049/ic.2013.0067, 2013.

[5] E. Alba and F. Chicano. Finding safety errors with ACO. In *9th International Conference on Genetic and Evolutionary Computation (GECCO'07)*, pages 1066–1073, 2007.

[6] E. Alba and F. Chicano. Searching for liveness property violations in concurrent systems with ACO. In *10th International Conference on Genetic and Evolutionary Computation (GECCO'08)*, pages 1727–1734, 2008.

[7] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2), 1994.

[8] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[9] J. Andrews, T. Menzies, and F. Li. Genetic algorithms for randomized unit testing. *IEEE Transactions on Software Engineering*, 37(1):80–94, 2011.

[10] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *33rd International Conference on Software Engineering (ICSE'11)*, pages 1–10, 2011.

[11] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model checking continuous-time Markov chains. *ACM Transactions on Computational Logic*, 1(1):162–170, 2000.

[12] C. Baier, B. Haverkort, H. Hermanns, and J. Katoen. Model checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, 2003.

[13] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[14] C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time markov chains. In *10th International Conference on Concurrency Theory (CONCUR'99)*, pages 146–161, 1999.

[15] C. Ballagny, N. Hameurlain, and F. Barbier. Mocas: A state-based component model for self-adaptation. In *3rd International Conference on Self-Adaptive and Self-Organizing Systems (SASO'09)*, pages 206–215, 2009.

[16] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[17] L. Baresi and C. Ghezzi. The disappearing boundary between development-time and run-time. In *Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER'10)*, pages 17–22, 2010.

[18] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks. UPPAAL 4.0. In *3rd International Conference on the Quantitative Evaluation of Systems (QEST'06)*, pages 125–126, 2006.

[19] N. Bencomo and G. Blair. Using architecture models to support the generation and operation of component-based adaptive systems. In *Software Engineering for Self-Adaptive Systems*, pages 183–200. 2009.

[20] M. Benjamin, H. Schmidt, P. Newman, and J. Leonard. Autonomy for unmanned marine vehicles with moos-ivp. In *Marine Robot Autonomy*, pages 47–90. 2013.

[21] M. R. Benjamin, H. Schmidt, P. M. Newman, and J. J. Leonard. Nested Autonomy for Unmanned Marine Vehicles with MOOS-IvP. *Journal of Field Robotics*, 27(6):834–875, 2010.

[22] A. Bianco and L. Alfaro. Model checking of probabilistic and nondeterministic systems. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 499–513. Springer, 1995.

[23] D. Bianculli, A. Filieri, C. Ghezzi, and D. Mandrioli. A syntactic-semantic approach to incremental verification. *CoRR*, abs/1304.8034, 2013.

[24] D. Bianculli, A. Filieri, C. Ghezzi, and D. Mandrioli. Syntactic-semantic incrementality for agile verification. *Science of Computer Programming*, 97, Part 1(0):47 – 54, 2015.

[25] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills, and Y. Diao. Able: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3):350–371, July 2002.

[26] R. Bloomfield and P. Bishop. Safety and assurance cases: Past, present and possible future — an Adelard perspective. In *Making Systems Safer*, pages 51–67. 2010.

[27] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems*, pages 48–70. 2009.

[28] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil. Deeco: An ensemble-based component system. In *16th International Symposium on Component-based Software Engineering (CBSE '13)*, pages 81–90, 2013.

[29] R. Calinescu. General-purpose autonomic computing. In *Autonomic Computing and Networking*, pages 3–30. 2009.

[30] R. Calinescu. Emerging techniques for the engineering of self-adaptive high-integrity software. In *Assurances for Self-Adaptive Systems*, volume 7740 of *LNCS*, pages 297–310. 2013.

[31] R. Calinescu, S. Gerasimou, and A. Banks. Self-adaptive software with decentralised control loops. In *18th International Conference on Fundamental Approaches to Software Engineering (FASE'15)*, pages 235–251, 2015.

[32] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Communications of the ACM*, 55(9):69–77, 2012.

[33] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic QoS management and optimization in service-based systems. *IEEE Transactions on Software Engineering*, 37(3):387–409, 2011.

[34] R. Calinescu, K. Johnson, and Y. Rafiq. Using observation ageing to improve Markovian model learning in QoS engineering. In *2nd International Conference on Performance Engineering (ICPE'11)*, pages 505–510, 2011.

[35] R. Calinescu, K. Johnson, and Y. Rafiq. Developing self-verifying service-based systems. In *28th International Conference on Automated Software Engineering (ASE'13)*, pages 734–737, 2013.

[36] R. Calinescu and S. Kikuchi. Formal methods @ runtime. In *16th Monterey Conference on Foundations of Computer Software: Modeling, Development, and Verification of Adaptive Systems (FOCS'10)*, volume 6662 of *LNCS*, pages 122–135. 2011.

[37] R. Calinescu, S. Kikuchi, and K. Johnson. Compositional reverification of probabilistic safety properties for large-scale complex it systems. In *Large-Scale Complex IT Systems. Development, Operation and Management*, volume 7539 of *LNCS*, pages 303–329. 2012.

[38] R. Calinescu and M. Kwiatkowska. Using quantitative analysis to implement autonomic IT systems. In *31st International Conference on Software Engineering (ICSE'09)*, pages 100–110, 2009.

[39] R. Calinescu, Y. Rafiq, K. Johnson, and M. E. Bakir. Adaptive model learning for continual verification of non-functional properties. In *5th International Conference on Performance Engineering (ICPE'14)*, pages 87–98, 2014.

[40] J. Cámara, R. de Lemos, C. Ghezzi, and A. Lopes, editors. *Assurances for Self-Adaptive Systems - Principles, Models, and Techniques*, volume 7740 of *LNCS*. Springer, 2013.

[41] J. Cámara, G. A. Moreno, and D. Garlan. Reasoning about human participation in self-adaptive systems. In *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'15)*, pages 146–156, 2015.

[42] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. An approach for QoS-aware service composition based on genetic algorithms. In *7th International Conference on Genetic and Evolutionary Computation (GECCO'05)*, pages 1069–1075, 2005.

[43] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *33rd International Symposium on Computer Architecture (ISCA'06)*, pages 264–276, 2006.

[44] T. Chen, E. M. Hahn, T. Han, M. Kwiatkowska, H. Qu, and L. Zhang. Model repair for markov decision processes. In *7th International Symposium on Theoretical Aspects of Software Engineering (TASE'13)*, pages 85–92, 2013.

[45] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller,

S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. Software engineering for self-adaptive systems: A research roadmap. pages 1–26. 2009.

[46] R. Cheung. A user-oriented software reliability model. *IEEE Transactions on Software Engineering*, SE-6(2):118–125, March 1980.

[47] S. L. Chung, S. Lafortune, and F. Lin. Limited lookahead policies in supervisory control of discrete event systems. *IEEE Transactions on Automatic Control*, 37:1921–1935, 1992.

[48] E. Clarke, W. Klieber, M. Novacek, and P. Zuliani. Model checking and the state explosion problem. In *Tools for Practical Software Verification*, volume 7682 of *LNCS*, pages 1–30. Springer, 2012.

[49] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, Apr. 1986.

[50] C. A. C. Coello, G. B. Lamont, and D. A. V. Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation)*. 2006.

[51] Z. Coker, D. Garlan, and C. Le Goues. SASS: Self-adaptation using stochastic search. In *10th International Symposium on Software Engineering for Adaptive and Self- Managing Systems (SEAMS'15)*, pages 168–174, 2015.

[52] R. Das, J. O. Kephart, C. Lefurgy, G. Tesauro, D. W. Levine, and H. Chan. Autonomic multi-agent management of power and performance in data centers. In *7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*, 2008.

[53] D. Dasgupta, G. Hernandez, A. Romero, D. Garrett, A. Kaushal, and J. Simien. On the use of informed initialization and extreme solutions sub-population in multi-objective evolutionary algorithms. In *IEEE Symposium on Computational Intelligence in Multi-Criteria Decision-Making (MCDM'09)*, pages 58–65, 2009.

[54] C. Daws. Symbolic and parametric model checking of discrete-time markov chains. In *First International Conference on Theoretical Aspects of Computing (ICTAC'04)*, volume 3407 of *LNCS*, pages 280–294. 2004.

[55] R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese. Software Engineering for Self-Adaptive Systems: Assurances (Dagstuhl Seminar 13511). *Dagstuhl Reports*, 3(12):67–96, 2014.

[56] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Goeschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, M. Litoiu, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, B. Schmerl, D. B. Smith, J. P. Sousa, G. Tamura, L. Tahvildari, N. M. Villegas, T. Vogel, D. Weyns, K. Wong, and J. Wuttke. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *LNCS*, pages 1–32. 2013.

[57] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[58] C. Dehnert, S. Junges, N. Jansen, F. Corzilius, M. Volk, H. Bruintjes, J. P. Katoen, and E. Ábrahám. PROPhESY: A PRObabilistic ParamEter SYnthesis Tool. In *27th International Conference on Computer Aided Verification (CAV'15)*, pages 214–231. Springer, 2015.

[59] E. Denney, I. Habli, and G. Pai. Dynamic safety cases for through-life safety assurance. In *37th International Conference on Software Engineering (ICSE'15)*, pages 587–590, 2015.

[60] P. J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, 1970.

[61] G. Di Marzo Serugendo, M.-P. Gleizes, and A. Karageorgos. Self-organization in multi-agent systems. *The Knowledge Engineering Review*, 20(2):165–189, June 2005.

[62] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3-4):313–341, 2008.

[63] N. D'Ippolito, V. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel. Hope for the best, prepare for the worst: Multi-tier control for adaptive systems. In *36th International Conference on Software Engineering (ICSE'14)*, pages 688–699, 2014.

[64] N. R. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesis of live behaviour models. In *18th International Symposium on Foundations of Software Engineering (FSE'10)*, pages 77–86, 2010.

[65] K. Draeger, V. Forejt, M. Kwiatkowska, D. Parker, and M. Ujma. Permissive controller synthesis for probabilistic systems. In *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *LNCS*, pages 531–546. 2014.

[66] J. J. Durillo and A. J. Nebro. jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42:760–771, 2011.

[67] M. Ehrgott. *Multicriteria Optimization*. 2005.

[68] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *31st International Conference on Software Engineering (ICSE'09)*, pages 111–121, 2009.

[69] K. Etessami, M. Kwiatkowska, M. Vardi, and M. Yannakakis. Multi-objective model checking of Markov decision processes. In *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *LNCS*, pages 50–65, 2007.

[70] European Commission. Horizon 2020: Work Programme 2016 - 2017. `http://ec.europa.eu/research/participants/data/ref/h2020/wp/2016_2017/main/h2020-wp1617-leit-ict_en.pdf`, July 2016.

[71] F. Faniyi and R. Bahsoon. Self-managing SLA compliance in cloud architectures: A market-based approach. In *3rd International Symposium on Architecting Critical Systems*, pages 61–70, 2012.

[72] L. Feng, M. Kwiatkowska, and D. Parker. Compositional verification of probabilistic systems using learning. In *7th International Conference on Quantitative Evaluation of Sysems (QEST'10)*, pages 133–142, 2010.

[73] L. Feng, M. Kwiatkowska, and D. Parker. Automated learning of probabilistic assumptions for compositional reasoning. In *Fundamental Approaches to Software Engineering (FASE'11)*, volume 6603 of *LNCS*, pages 2–17. Springer, 2011.

[74] F. Ferrucci, M. Harman, J. Ren, and F. Sarro. Not going to take this anymore: Multi-objective overtime planning for software engineering projects. In *35th International Conference on Software Engineering (ICSE'13)*, pages 462–471, 2013.

[75] A. Filieri and C. Ghezzi. Further steps towards efficient runtime verification: Handling probabilistic cost models. In *First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA'12)*, pages 2–8, 2012.

[76] A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. In *33rd International Conference on Software Engineering (ICSE'11)*, pages 341–350, 2011.

[77] A. Filieri, C. Ghezzi, and G. Tamburrelli. A formal approach to adaptive software: continuous assurance of non-functional requirements. *Formal Aspects of Computing*, 24(2):163–186, 2012.

[78] A. Filieri, H. Hoffmann, and M. Maggio. Automated design of self-adaptive software with control- theoretical formal guarantees. In *36th International Conference on Software Engineering (ICSE'14)*, pages 299–310, 2014.

[79] A. Filieri and G. Tamburrelli. Probabilistic verification at runtime for self-adaptive systems. In *Assurances for Self-Adaptive Systems*, volume 7740 of *LNCS*, pages 30–59. 2013.

[80] M. Fisher, L. Dennis, and M. Webster. Verifying autonomous systems. *Communications of the ACM*, 56(9):84–93, Sept. 2013.

[81] V. Forejt, M. Kwiatkowska, G. Norman, D. Parker, and H. Qu. Quantitative multi-objective verification for probabilistic systems. In *17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11)*, volume 6605 of *LNCS*, pages 112–127, 2011.

[82] V. Forejt, M. Kwiatkowska, D. Parker, H. Qu, and M. Ujma. Incremental runtime verification of probabilistic systems. In *3rd International Conference on Runtime Verification (RV'12)*, volume 7687 of *LNCS*, pages 314–319. Springer, 2012.

[83] V. Forejt, M. Kwiatkowska, D. Parker, H. Qu, and M. Ujma. Incremental runtime verification of probabilistic systems. Technical Report RR-12-05, Department of Computer Science, University of Oxford, 2012.

[84] F. Fouquet, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J.-M. Jézéquel. Kevoree modeling framework (KMF): Efficient modeling techniques for runtime use. *CoRR*, abs/1405.6817, 2014.

[85] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *Fifth International Conference on Software Testing, Verification and Validation (ICST'12)*, pages 121–130, 2012.

[86] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[87] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *31st International Symposium on Security and Privacy (SP'10)*, pages 45–60, 2010.

[88] T. Friedrich and M. Wagner. Seeding the initial population of multi-objective evolutionary algorithms: A computational study. *Applied Soft Computing*, 33:223 – 230, 2015.

[89] S. Gallotti, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Quality prediction of service compositions through probabilistic model checking. In *4th International Conference on Quality of Software-Architectures: Models and Architectures (QoSA'08)*, volume 5281 of *LNCS*, pages 119–134. 2008.

[90] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *18th International Conference on World Wide Web (WWW '09)*, pages 431–440, 2009.

[91] A. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[92] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, Oct 2004.

[93] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *First Workshop on Self-healing Systems (WOSS '02)*, pages 33–38, 2002.

[94] S. Gerasimou, R. Calinescu, and A. Banks. Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration. In *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*, pages 115–124, 2014.

[95] C. Ghezzi. Evolution, adaptation, and the quest for incrementality. In *Large-Scale Complex IT Systems. Development, Operation and Management*, volume 7539 of *LNCS*, pages 369–379. 2012.

[96] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *31st International Conference on Software Engineering (ICSE'09)*, pages 430–440, 2009.

[97] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli. Mining behavior models from user-intensive web applications. In *36th International Conference on Software Engineering (ICSE'14)*, pages 277–287, 2014.

[98] C. Ghezzi, L. S. Pinto, P. Spoletini, and G. Tamburrelli. Managing non-functional uncertainty via model-driven adaptivity. In *35th International Conference on Software Engineering (ICSE'13)*, pages 33–42, 2013.

[99] D. Gil de La Iglesia and D. Weyns. MAPE-K formal templates to rigorously design behaviors for self-adaptive systems. *ACM Transactions on Autonomous and Adaptive Systems*, 2015.

[100] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *10th International Symposium on Computer Architecture (ISCA'83)*, pages 124–131, 1983.

[101] V. Grassi, M. Marzolla, and R. Mirandola. Qos-aware fully decentralized service assembly. In *8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'13)*, pages 53–62, 2013.

[102] V. Grassi, R. Mirandola, and A. Sabetta. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *Journal of Systems and Software*, 80(4):528 – 558, 2007.

[103] J. J. Grefenstette. Incorporating problem specific knowledge into genetic algorithms. *Genetic algorithms and simulated annealing*, pages 42–60, 1987.

[104] L. Grunske. Specification patterns for probabilistic quality properties. In *30th International Conference on Software Engineering (ICSE'08)*, pages 31–40, 2008.

[105] GSN Working Group Online. Goal structuring notation standard, version 1, November 2011.

[106] A. Haber, H. Rendel, B. Rumpe, and I. Schaefer. Evolving delta-oriented software product line architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management: 17th Monterey Workshop*, pages 183–208, 2012.

[107] E. Hahn, H. Hermanns, and L. Zhang. Probabilistic reachability for parametric markov models. *International Journal on Software Tools for Technology Transfer*, 13(1):3–19, 2011.

[108] E. M. Hahn, T. Han, and L. Zhang. Synthesis for PCTL in parametric Markov decision processes. In *3rd international conference on NASA Formal methods (NFM'11)*, pages 146–161, 2011.

[109] E. M. Hahn, H. Hermanns, B. Wachter, and L. Zhang. PARAM: A model checker for parametric Markov models. In *22nd International Conference on Computer Aided Verification (CAV'10)*, volume 6174 of *LNCS*, pages 660–664. 2010.

[110] E. M. Hahn, H. Hermanns, and L. Zhang. Probabilistic reachability for parametric Markov models. In *16th International SPIN Workshop on Model Checking Software (SPIN'09)*, pages 88–106, 2009.

[111] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[112] M. Harman, E. Burke, J. Clark, and X. Yao. Dynamic adaptive search based software engineering. In *6th International Symposium on Empirical Software Engineering and Measurement (ESEM'12)*, pages 1–8, 2012.

[113] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang. Search based software engineering for software product line engineering: A survey and directions for future work. In *18th International Software Product Line Conference*, pages 5–18, 2014.

[114] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*, pages 212–222, 2011.

[115] M. Harman, Y. Jia, W. B. Langdon, J. Petke, I. H. Moghadam, S. Yoo, and F. Wu. Genetic improvement for adaptive software engineering. In *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*, pages 1–4, 2014.

[116] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11:1–11:61, 2012.

[117] M. Harman, P. McMinn, J. de Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical Software Engineering and Verification*, volume 7007 of *LNCS*, pages 1–59. 2012.

[118] I. Hatzakis and D. Wallace. Dynamic multi-objective optimization with evolutionary algorithms: A forward-looking approach. In *8th International Conference on Genetic and Evolutionary Computation (GECCO'06)*, pages 1201–1208, 2006.

[119] R. Hawkins, K. Clegg, R. Alexander, and T. Kelly. Using a software safety argument pattern catalogue: Two case studies. In *Computer Safety, Reliability, and Security*, pages 185–198. 2011.

[120] R. Hawkins, I. Habli, and T. Kelly. Principled construction of software safety cases. In *SAFECOMP 2013-SASSUR Workshop*, 2013.

[121] R. Hawkins, I. Habli, and T. Kelly. The principles of software safety assurance. In *31st International System Safety Conference*, 2013.

[122] R. Hawkins, I. Habli, T. Kelly, and J. McDermid. Assurance cases and prescriptive software safety certification: A comparative study. *Safety Science*, 59:55–71, 2013.

[123] S. Helwig and R. Wanka. Theoretical analysis of initial particle swarm behavior. In *10th International Conference on Parallel Problem Solving from Nature (PPSN'08)*, pages 889–898, 2008.

[124] A. G. Hernandez-Diaz, C. A. C. Coello, F. Perez, R. Caballero, J. Molina, and L. V. Santana-Quintero. Seeding the initial population of a multi-objective evolutionary algorithm using gradient-based information. In *IEEE Congress on Evolutionary Computation*, pages 1617–1624, 2008.

[125] C. A. R. Hoare. An axiomatic basis for computer programming. *Commununications of the ACM*, 12(10):576–580, Oct. 1969.

[126] P. Horn. Autonomic computing: IBM's perspective on the state of information technology, 2001.

[127] M. C. Huebscher and J. A. McCann. A survey of autonomic computing-degrees, models, and applications. *ACM Computunig Surveys*, 40(3):7:1–7:28, Aug. 2008.

[128] M. U. Iftikhar and D. Weyns. Activforms: Active formal models for self-adaptation. In *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*, pages 125–134, 2014.

[129] C. Johnson. Genetic programming with fitness based on model checking. In *Genetic Programming*, volume 4445 of *LNCS*, pages 114–124. 2007.

[130] K. Johnson, R. Calinescu, and S. Kikuchi. An incremental verification framework for component-based software systems. In *16th International Symposium on Component-based Software Engineering (CBSE'13)*, pages 33–42, 2013.

[131] T. Kalmar-Nagy, R. D'Andrea, and P. Ganguly. Near-optimal dynamic trajectory generation and control of an omnidirectional vehicle. *Robotics and Autonomous Systems*, 46(1):47 – 64, 2004.

[132] J. P. Katoen, M. Khattri, and I. S. Zapreev. A Markov reward model checker. In *2nd International Conference on Quantitative Evaluation of Systems (QEST'05)*, pages 243–244, 2005.

[133] J. P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker MRMC. *Performance Evaluation*, 68(2):90 – 104, 2011.

[134] G. Katz and D. Peled. Synthesis of parametric programs using genetic programming and model checking. In *15th International Workshop on Verification of Infinite-State Systems (INFINITY'13)*, pages 70–84, 2013.

[135] B. Kazimipour, X. Li, and A. K. Qin. A review of population initialization techniques for evolutionary algorithms. In *IEEE Congress on Evolutionary Computation (CEC'14)*, pages 2585–2592, 2014.

[136] B. Kazimipour, X. Li, and A. K. Qin. Why advanced population initialization techniques perform poorly in high dimension? In *10th International Conference on Simulated Evolution and Learning*, pages 479–490, 2014.

[137] H. Kellerer, U. Pferschy, and D. Pisinger. The multiple-choice knapsack problem. In *Knapsack Problems*, pages 317–347. 2004.

[138] T. Kelly and R. Weaver. The Goal Structuring Notation – a safety argument notation. In *Assurance Cases Workshop*, 2004.

[139] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[140] C. Klein, M. Maggio, K.-E. Arzén, and F. Hernández-Rodriguez. Brownout: Building more robust cloud applications. In *36th International Conference on Software Engineering (ICSE'14)*, pages 700–711, 2014.

[141] A. Komuravelli, C. S. Pasareanu, and E. M. Clarke. Learning probabilistic systems from tree samples. In *27th International Symposium on Logic in Computer Science (LICS'12)*, pages 441–450, 2012.

[142] R. Kota, N. Gibbins, and N. R. Jennings. Decentralized approaches for self-adaptation in agent organizations. *ACM Transactions on Autonomous and Adaptive Systems*, 7(1):1:1–1:28, May 2012.

[143] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, 17(PB):184–206, Feb. 2015.

[144] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang. Power and performance management of virtualized computing environments via lookahead control. In *5th International Conference on Autonomic Computing (ICAC'08)*, pages 3–12, 2008.

[145] M. Kwiatkowska. Quantitative verification: models, techniques and tools. In *6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers (ESEC-FSE'07)*, pages 449–458, 2007.

[146] M. Kwiatkowska. From software verification to 'everyware' verification. *Computer Science - Research and Development*, pages 295–310, 2013.

[147] M. Kwiatkowska, G. Norman, and D. Parker. Quantitative analysis with the probabilistic model checker PRISM. *Electronic Notes in Theoretical Computer Science*, 153(2):5 – 31, 2006.

[148] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, pages 220–270. Springer, 2007.

[149] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: verification of probabilistic real-time systems. In *23rd International Conference on Computer Aided Verification (CAV'11)*, pages 585–591. Springer, 2011.

[150] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic verification of Herman's self-stabilisation algorithm. *Formal Aspects of Computing*, 24(4-6):661–670, 2012.

[151] M. Kwiatkowska, G. Norman, D. Parker, and H. Qu. Assume-guarantee verification for probabilistic systems. In *16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, volume 6015 of *LNCS*, pages 23–37. 2010.

[152] M. Kwiatkowska, G. Norman, and R. Segala. Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM. In *13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 194–206. 2001.

[153] M. Kwiatkowska and D. Parker. Automated verification and strategy synthesis for probabilistic systems. In *11th International Symposium on Automated Technology for Verification and Analysis (ATVA'13)*, volume 8172 of *LNCS*, pages 5–22, 2013.

[154] M. Kwiatkowska, D. Parker, and H. Qu. Incremental quantitative verification for Markov decision processes. In *41st International Conference on Dependable Systems Networks (DSN'11)*, pages 359–370, 2011.

[155] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[156] J. Manyika, M. Chui, J. Bughin, R. Dobbs, P. Bisson, and A. Marrs. Disruptive technologies: Advances that will transform life, business, and the global economy. `http://www.mckinsey.com/business-functions/business-technology/our-insights/disruptive-technologies`, May 2013.

[157] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., 1990.

[158] I. Meedeniya and L. Grunske. An efficient method for architecture-based reliability evaluation for evolving systems with changing parameters. In *21st International Symposium on Software Reliability Engineering (ISSRE'10)*, pages 229–238, 2010.

[159] N. Megiddo and D. S. Modha. Outperforming LRU with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, 2004.

[160] L. L. Minku and X. Yao. Software effort estimation as a multiobjective learning problem. *ACM Transactions on Software Engineering and Methodology*, 22(4):35:1–35:32, 2013.

[161] V. Mirrokni, N. Thain, and A. Vetta. A theoretical examination of practical game playing: Lookahead search. In *5th International Conference on Algorithmic Game Theory (SAGT'12)*, pages 251–262, 2012.

[162] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl. Proactive self-adaptation under uncertainty: A probabilistic model checking approach. In *10th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*, pages 1–12, 2015.

[163] V. Nallur and R. Bahsoon. A decentralized self-adaptation mechanism for service-based applications in the cloud. *IEEE Transactions on Software Engineering*, 39(5):591–612, 2013.

[164] D. S. Nau. Decision quality as a function of search depth on game trees. *Journal of the ACM*, 30(4):687–708, 1983.

[165] D. S. Nau, M. Lustrek, A. Parker, I. Bratko, and M. Gams. When is it better not to look ahead? *Artificial Intelligence*, 174(16-17):1323 – 1338, 2010.

[166] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba. MOCell: A cellular genetic algorithm for multiobjective optimization. *International Journal of Intelligent Systems*, 24(7):726–746, 2009.

[167] N. Nostro, R. Spalazzese, F. D. Giandomenico, and P. Inverardi. Achieving functional and non functional interoperability through synthesized connectors. *Journal of Systems and Software*, 111(C):185–199, Jan. 2016.

[168] S. Oman and P. Cunningham. Using case retrieval to seed genetic algorithms. *International Journal of Computational Intelligence and Applications*, 01(01):71–82, 2001.

[169] P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and their Applications*, 14(3):54–62, 1999.

[170] S. Ortmanns, H. Ney, and A. Eiden. Language-model look-ahead for large vocabulary speech recognition. In *4th International Conference on Spoken Language (ICSLP'96)*, volume 4, pages 2095–2098 vol.4, 1996.

[171] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series*, pages 123–144. Springer, 1985.

[172] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Computing Surveys*, 35(4):374–398, 2003.

[173] K. Praditwong, M. Harman, and X. Yao. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, 37(2):264–282, March 2011.

[174] Q. Qiu, Q. Qu, and M. Pedram. Stochastic modeling of a power-managed system-construction and optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(10):1200–1217, 2001.

[175] A. Ramirez, D. Knoester, B. Cheng, and P. McKinley. Plato: a genetic algorithm approach to run-time reconfiguration in autonomic computing systems. *Cluster Computing*, 14(3):229–244, 2011.

[176] J. Rao, X. Bu, C.-Z. Xu, and K. Wang. A distributed self-learning approach for elastic provisioning of virtualized cloud resources. In *19th International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS'11)*, pages 45–54, 2011.

[177] S. Redfield. Cooperation between underwater vehicles. In *Marine Robot Autonomy*, pages 257–286. 2013.

[178] J. Ren, M. Harman, and M. Di Penta. Cooperative co-evolutionary optimization of software project staff assignments and job scheduling. In *3rd International Symposium on Search Based Software Engineering (SSBSE'11)*, volume 6956 of *LNCS*, pages 127–141. 2011.

[179] Robotics & Autonomous Systems Special Interest Group. RAS 2020: Robotics and Autonomous Systems. `https://connect.innovateuk.org/documents/2903012/16074728/RASUKStrategy`, July 2014.

[180] S. M. Ross. *Stochastic Processes*. Wiley, 2 edition, 1995.

[181] Royal Academy of Engineering. Establishing High-Level Evidence for the Safety and Efficacy of Medical Devices and Systems, January 2013.

[182] J. Rutten, M. Kwiatkowska, G. Norman, and D. Parker. *Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems*, volume 23 of *CRM Monograph Series*. American Mathematical Society, 2004.

[183] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):14:1–14:42, 2009.

[184] U. K. Sarkar, P. P. Chakrabarti, S. Ghose, and S. C. DeSarkar. Improving greedy algorithms by lookahead-search. *Journal of Algorithms*, 16(1):1–23, 1994.

[185] A. Sayyad, J. Ingram, T. Menzies, and H. Ammar. Scalable product line configuration: A straw to break the camel's back. In *28th International Conference on Automated Software Engineering (ASE'13)*, pages 465–474, 2013.

[186] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *14th International Conference on Software Product Lines (SPLC'10)*, pages 77–91, 2010.

[187] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995.

[188] A. Sesic, S. Dautovic, and V. Malbasa. Dynamic power management of a system with a two-priority request queue using probabilistic-model checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(2):403–407, 2008.

[189] M. Seto, L. Paull, and S. Saeedi. Introduction to autonomy for marine robots. In *Marine Robot Autonomy*, pages 1–46. 2013.

[190] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[191] I. Sommerville, D. Cliff, R. Calinescu, J. Keen, T. Kelly, M. Kwiatkowska, J. Mcdermid, and R. Paige. Large-Scale Complex IT Systems. *Communications of the ACM*, 55(7):71–77, July 2012.

[192] J. Spriggs. *GSN – The Goal Structuring Notation. A Structured Approach to Presenting Arguments.* Springer, 2012.

[193] A. Sridharan, R. Guérin, and C. Diot. Achieving near-optimal traffic engineering solutions for current OSPF/IS-IS networks. *IEEE/ACM Transactions on Networking*, 13(2):234–247, 2005.

[194] C. Stylianou, S. Gerasimou, and A. Andreou. A novel prototype tool for intelligent software project scheduling and staffing enhanced with personality factors. In *24th International Conference on Tools with Artificial Intelligence (ICTAI'12)*, pages 277–284, 2012.

[195] D. Sykes, J. Magee, and J. Kramer. Flashmob: Distributed adaptive self-assembly. In *6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'11)*, pages 100–109, 2011.

[196] G. Tamura and et al. Towards practical runtime verification and validation of self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *LNCS*. 2013.

[197] G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart, and S. R. White. A multi-agent systems approach to autonomic computing. In *3rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS'04)*, pages 464–471, 2004.

[198] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *3rd International Conference on Autonomic Computing (ICAC'06)*, pages 65–73, June 2006.

[199] UK Health & Safety Commission. The use of computers in safety-critical applications, 1998.

[200] UK Ministry of Defence. Defence Standard 00-56, Issue 4: Safety Management Requirements for Defence Systems, June 2007.

[201] A. Ulusoy, T. Wongpiromsarn, and C. Belta. Incremental controller synthesis in probabilistic environments with temporal logic constraints. *International Journal on Robotic Research*, 33(8):1130–1144, 2014.

[202] D. A. Van Veldhuizen. *Multiobjective Evolutionary Algorithms: Classifications, Analyses, and New Innovations*. PhD thesis, 1999.

[203] A. Vargha and H. D. Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal on Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[204] V. V. Vazirani. *Approximation Algorithms*. Springer, New York, USA, 2001.

[205] D. Weyns, N. Bencomo, R. Calinescu, J. Cámara, C. Ghezzi, V. Grassi, L. Grunske, P. Inverardi, J.-M. Jezequel, S. Malek, R. Mirandola, M. Mori, and G. Tamburrelli. Perpetual assurances in self-adaptive systems. In *Software Engineering for Self-Adaptive Systems IV, LNCS*. Springer, 2016.

[206] D. Weyns and R. Calinescu. Tele assistance: A self-adaptive service-based system exemplar. In *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'15)*, pages 88–92, May 2015.

[207] D. Weyns, R. Haesevoets, A. Helleboogh, T. Holvoet, and W. Joosen. The MACODO middleware for context-driven dynamic agent organizations. *ACM Transactions on Autonomous and Adaptive Systes*, 5(1):3:1–3:28, 2010.

[208] D. Weyns, M. U. Iftikhar, D. G. de la Iglesia, and T. Ahmad. A survey of formal methods in self-adaptive systems. In *5th International C* Conference on Computer Science and Software Engineering (C3S2E'12)*, pages 67–79, 2012.

[209] J. White, B. Dougherty, and D. C. Schmidt. Selecting highly optimal architectural feature sets with filtered Cartesian flattening. *Journal of Systems and Software*, 82(8):1268–1284, 2009.

[210] M. Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2009.

[211] M. Wooldridge and N. R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10:115–152, 6 1995.

[212] C. Ye, S. Cheung, and W. Chan. Process evolution with atomicity consistency. In *2nd International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'07)*, pages 19–19, 2007.

[213] L. Yin and G. Cao. Supporting cooperative caching in ad hoc networks. *IEEE Transactions on Mobile Computing*, 5(1):77–89, 2006.

[214] H. L. S. Younes. Ymer: A statistical model checker. In *17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 429–433. 2005.

[215] C.-H. Yu. *Biologically-inspired Control for Self-adaptive Multiagent Systems*. PhD thesis, Harvard University, Cambridge, MA, USA, 2010. AAI3415434.

[216] F. Zambonelli and M. Viroli. A survey on nature-inspired metaphors for pervasive service ecosystems. *International Journal of Pervasive Computing and Communications*, 7(3):186–204, 2011.

[217] J. Zhang and B. H. Cheng. Model-based development of dynamically adaptive software. In *28th International Conference on Software Engineering (ICSE'06)*, pages 371–380, 2006.

[218] J. Zhang and B. H. Cheng. Using temporal logic to specify adaptive program semantics. *Journal of Systems and Software*, 79(10):1361 – 1369, 2006.

[219] E. Zitzler, D. Brockhoff, and L. Thiele. The hypervolume indicator revisited: On the design of Pareto-compliant indicators via weighted integration. In *4th International Conference on Evolutionary Multi-criterion Optimization (EMO'07)*, pages 862–876, 2007.

[220] E. Zitzler, J. Knowles, and L. Thiele. Quality assessment of Pareto set approximations. In *Multiobjective Optimization*, volume 5252 of *LNCS*, pages 373–404. 2008.

[221] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength Pareto evolutionary algorithm. In *Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems (EUROGEN'01)*, pages 95–100, 2001.

[222] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.

[223] E. Zitzler, L. Thiele, M. Laumanns, C. Fonseca, and V. da Fonseca. Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132, 2003.