A Control-based Approach for
Self-adaptive Software Systems
with Formal Guarantees

Licentiate Thesis
Computer Science
Advisor: Danny Weyns
Co-advisor: Martina Maggio
2017
A thesis for the Degree of Licentiate of Philosophy in Computer Science

A Control-based Approach for Self-adaptive Software Systems with Formal Guarantees

Stepan Shevtsov
e-mail: stepan.shevtsov@lnu.se, stepan.shevtsov@kuleuven.be

Linnaeus University
Faculty of Technology
http://www.lnu.se/
Växjö, Sweden, 2017

Katholieke Universiteit Leuven
Faculty of Engineering Science
http://www.kuleuven.be
Acknowledgments

The research presented in this Thesis would not have been possible without the help of a number of people. Here, I would like to mention those who contributed the most.

I would like to specially thank my supervisor, professor Danny Weyns, for his continuous support throughout the years of my PhD, his invaluable input to the progress of my research and for sharing his large experience. I am sure that I became not only a better researcher but a better person thanks to Danny.

I would like to thank my co-supervisor, associate professor Martina Maggio. Her expertise in control theory greatly increased the quality of our publications and helped me to come up with the approach described in this thesis.

I would like to thank my colleague, Muhammad Usman Iftikhar, who worked on many technical projects that were used in my research. In addition to that, our discussions with Usman helped in developing my research ideas.

I would like to thank assistant professor Antonio Filieri for being an opponent for this Thesis and for providing feedback that I will use to improve this work.

I would like to thank the Swedish Research Council and the Faculty of Technology of Linnaeus University for their financial support. All the conference trips and research activities would not be possible without it. I would also like to thank KU Leuven for providing opportunities for research cooperation.

I would like to thank my mother, Anna. She is the person that always gets worried when I travel abroad, but at the same time gets happy when I succeed. She always supported me and gave precious advices. For example, she helped me to come up with the idea of Simplex, which I continuously used in this Thesis.

Finally, I would like to thank my grandparents, Alla Ilyinih and professor Ziya Memedlyaev, who shared their research and life experience, which helped in many situation throughout the years of my research.

Växjö, Sweden
October 25th, 2017
### Contents

1 Introduction
   1.1 Problem Definition .................................. 2
   1.2 Scientific Approach ................................ 3
   1.3 Scope of Research ................................ 4
   1.4 Contributions .................................... 7
   1.5 Thesis Overview .................................. 7

2 Research Approach
   2.1 Systematic Literature Review ......................... 9
   2.2 Analytical Method ................................ 11
   2.3 Informal Exploratory Case Studies ................... 12

3 Research Background
   3.1 Self-adaptive Software Systems ...................... 13
   3.2 Control theory and software systems ............... 15
   3.3 Push-Button Methodology ........................... 17
   3.4 Simplex ........................................... 18
   3.5 Formal Guarantees ................................ 19

4 A Systematic Literature Review on Control-Based Software Adaptation 23
   4.1 Introduction ...................................... 24
   4.2 Focus of the Literature Study ....................... 26
   4.3 Self-adaptive Systems Background ................... 29
   4.4 Control Theory Background ........................ 31
   4.5 Related Efforts ................................... 34
   4.6 Research Method .................................. 36
   4.7 Result Analysis ................................... 46
   4.8 Discussion ....................................... 67
   4.9 Threats to validity ................................ 71
   4.10 Conclusion ...................................... 73

5 SimCA: a Control-Based Approach to Adapt Software Systems 75
   5.1 Introduction ...................................... 76
   5.2 Motivating scenario: UUV System .................... 78
   5.3 Simplex Control Adaptation ........................ 80
   5.4 Evaluation of Guarantees ............................ 86
   5.5 Experimental Evaluation ............................ 89
   5.6 Related Work ...................................... 98
   5.7 Conclusions ....................................... 100

6 SimCA vs ActivFORMS: comparative analysis 101
   6.1 Introduction ...................................... 102
   6.2 Adaptation Scenario: TAS .......................... 103
   6.3 Studied Approaches ................................. 105
   6.4 Comparative Evaluation ............................. 109
   6.5 Discussion ........................................ 117
   6.6 Conclusions ....................................... 118
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>SimCA*: a Control-Based Approach to Handle New and Changing Requirements</td>
<td>119</td>
</tr>
<tr>
<td></td>
<td>7.1 Introduction</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>7.2 State of the art overview</td>
<td>122</td>
</tr>
<tr>
<td></td>
<td>7.3 Problem Definition</td>
<td>122</td>
</tr>
<tr>
<td></td>
<td>7.4 Simplex Control Adaptation* - SimCA*</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>7.5 Formal Evaluation of Guarantees</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>7.6 Experimental Evaluation</td>
<td>132</td>
</tr>
<tr>
<td></td>
<td>7.7 Conclusions</td>
<td>140</td>
</tr>
<tr>
<td>8</td>
<td>Conclusions</td>
<td>145</td>
</tr>
<tr>
<td></td>
<td>8.1 Contributions</td>
<td>145</td>
</tr>
<tr>
<td></td>
<td>8.2 Future Work</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>8.3 Closing Reflections</td>
<td>148</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>151</td>
</tr>
</tbody>
</table>
List of Figures

2.1 The detailed scheme of conducted research .................................. 9
2.2 The scheme of a systematic literature review .............................. 10
2.3 The analytical method .......................................................... 11
2.4 The exploratory case study method ......................................... 12
3.1 Architecture-based self-adaptive software system ....................... 14
3.2 Control-based self-adaptive software system ............................ 15
3.3 The Push-Button Methodology ............................................... 17
3.4 Solving tasks with simplex: a geometric representation ............. 19
3.5 The standard control-theoretical guarantees ............................ 20
4.1 The three layers representing modern computing systems. The focus of the review is displayed in grey. ................................. 27
4.2 Control-theoretical techniques. In grey the techniques that are considered in this literature review. ................................. 28
4.3 Model of a self-adaptive software system ................................ 30
4.4 Block diagram of a feedback control scheme ........................... 32
4.5 Block diagram of a feedforward control scheme ....................... 33
4.6 Control-Theoretical Guarantees ............................................. 46
4.7 F2: Number of analyzed primary studies by year ....................... 47
4.8 F6: Presentation quality scores .............................................. 48
4.9 Results for data items required to answer RQ1 .......................... 50
4.10 Results for different properties of system models ..................... 52
4.11 F14.1: Sensors ................................................................. 55
4.12 F14.2: Actuators ............................................................... 58
4.13 Relation between F14.1 Sensors and F13 System model ............ 60
4.14 Relation between F14.2 Actuators and F13 System model .......... 61
4.15 Results for different properties of system models ..................... 62
4.16 Relation between F16 Controller type and F13 System model .... 63
4.17 F19 Software qualities ....................................................... 63
4.18 F16: Controller type vs F19: Software qualities ....................... 67
5.1 Identification phase of SimCA ................................................. 81
5.2 Control Synthesis phase of SimCA ......................................... 83
5.3 Operation phase of SimCA .................................................... 85
5.4 Properties guaranteed by the controllers ................................ 86
5.5 UUV adaptation with runtime changes ................................... 90
5.6 The influence of controller pole on SimCA ............................. 92
5.7 SimCA vs Simplex: constant disturbance ............................... 93
5.8 Detection of Infeasible Solution in SimCA ............................... 93
5.9 SimCA with 25 sensor combinations .................................... 94
5.10 TAS workflow ................................................................. 95
5.11 SimCA on TAS scenario ..................................................... 96
6.1 TAS workflow ................................................................. 103
6.2 A self-adaptive software with SimCA .................................... 105
Chapter 1

Introduction

Nowadays, software systems are widely used to support or even replace humans in performing different tasks. A possibility to create software being able to function without a human intervention in a continuously changing environment could not be imagined even a couple of decades ago. It is truly fascinating that today such systems are represented by a number of software applications used across different domains. Some examples are unmanned underwater vehicles used for oceanic surveillance, e-health systems that provide medical support to patients, smart homes, and Internet of Things networks.

These systems have some characteristics in common: they have multiple (possibly conflicting and changing) requirements; the requirements are strict (e.g., an attempt to save energy should not lead to a loss of an expensive underwater vehicle during a mission or an e-health system crash); these systems are expected to deal with changing availability of resources, cope with failures, and autonomously adapt to changes in the dynamic execution environment. Engineering such systems is a complex task due to different kinds of uncertainties that are difficult to take into account at design time [139]. For example, a discharged battery in the patient health tracker or a rainy weather that disrupts outdoor communications in the Internet of Things network may lead to catastrophic consequences.

More than a decade ago, a seminal paper of Kephart and Chess [93] stated that the only way to cope with the growing complexity of engineering computing systems is by the means of autonomic computing — a system that manages itself autonomously given the high-level goals. This concept has led to the development of a whole new class of systems that are known today as self-adaptive systems. The idea of self-adaptation is to autonomously handle different kinds of uncertainties during system operation, when the knowledge to deal with those uncertainties becomes available [42, 49]. To that end, the system is equipped with a feedback loop that monitors the system in its environment, and adapts the system’s parameters or configuration to ensure that the requirements are met under changing conditions.

However, strict requirements of many software systems, such as the examples given above, cannot be addressed by a basic feedback loop. Those systems require guarantees for the adaptation results, such as system stability, robustness to disturbances and others [35, 41, 142]. Driven by the need for guarantees, researchers started applying control theory to design self-adaptive systems. Control theory is an established formal discipline typically used to control the behavior of physical systems, such as production lines, aircrafts, etc. Starting from 2000’s, a
number of research efforts studied the interplay between control theory and adaptation of computing systems [126, 111], a notable one being the book of Hellerstein et al. [79]. However, most of these efforts concentrated on resource allocation or admission control solutions, which is fundamentally different from adaptation of software systems [48, 22, 65, 139].

Control-based software adaptation (CBSA) only recently emerged as a research field [48]. Consequently, CBSA research is not systematized and it is lacking solutions for adaptation problems posed by the systems with strict requirements. Even though some of these solutions exist [63, 129, 98], they are designed for a specific application with little possibility of reuse, meaning that new problems require new dedicated solutions.

The aim of this Thesis is to systematize knowledge in the CBSA research field and to produce a reusable control-based approach to design self-adaptive software systems that provides guarantees on the adaptation results (hence, allowing to engineer systems with strict requirements).

1.1 Problem Definition

From the characteristics of systems with strict requirements (discussed above), the following generic adaptation problem can be identified:

*To satisfy multiple stakeholder requirements in the presence of disturbances and goal changes, and provide guarantees on the adaptation results.*

Throughout this dissertation, with *disturbances* we mean changes in the environment, inaccurate measurements and component failures. With *goal changes* we mean addition or removal of system requirements, or change of requirement values at runtime.\(^1\)

In order to make the definition of the adaptation problem more precise, we determined the types of stakeholder requirements by looking at existing self-adaptive artifacts\(^2\) and systems with strict requirements from the literature, e.g. [65, 98]. We identified three commonly used types of stakeholder requirements: setpoint requirements (*S-reqs*), where a certain system property should be maintained at a desired level, threshold requirements (*T-reqs*) that keeps a value above/below a threshold, and optimization requirements (*O-reqs*), where certain property should be minimized or maximized. For example, [98] defines a particular response time of a web server that the adaptation solution should guarantee (*S-req*). The automated traffic routing system [160] includes vehicles that should not surpass the speed limit (*T-req*) and should minimize the travel time to a destination point (*O-req*). Another example, Znn.com [73], features a website that should serve news content to the customers within a specified response time (*T-req*) while minimizing the operation cost (*O-req*). In this thesis, we will refer to a combination of *S-reqs*, *T-reqs* and *O-reqs* as *STO-reqs*.

\(^1\)A requirement specifies what is expected from a system by the stakeholder. A goal specifies what should be achieved by a system at a particular moment during operation; a goal is typically calculated based on requirements.

\(^2\)http://self-adaptive.org
1.2 Scientific Approach

Based on these requirement types and with reusability in mind, we refined the generic adaptation problem inherent to systems with strict requirements into the following research problem \( RP \):

**RP:** _To guarantee the satisfaction of STO-reqs with a reusable CBSA solution in the presence of disturbances and goal changes._

We identified three core research questions from this problem. As there is no systematic study consolidating knowledge in CBSA, with research question \( RQ_1 \) we want to get trends of research in CBSA and study existing CBSA approaches that may potentially address the research problem \( RP \):

**RQ1:** “What CBSA approaches are used to deal with STO-reqs in the existing literature?”

Though existing control-based approaches have different implementations, they include two key components. They create a mathematical model of software and use a control solution that adapts software based on this model in order to satisfy the system requirements. Therefore, with research question \( RQ_2 \) we want to identify the two key components of a CBSA approach that can address the research problem \( RP \):

**RQ2:** “What are the appropriate models and control solutions that can be used to address STO-reqs, and deal with disturbances and goal changes?”

Systems with strict requirements prioritize different software qualities. While for an unmanned underwater vehicle the key qualities are accuracy of measurements and efficiency in consuming available energy, the medical assistance system prioritizes high reliability and performance. Those systems also require different guarantees: a underwater vehicle functions in a highly disturbed environment and requires highest robustness in the first place, while the medical system works with disturbances of much lower magnitude, so it may prioritize settling time in order to increase system responsiveness. Hence, with research question \( RQ_3 \) we want to obtain an overview of types of goals that can be addressed and types of guarantees that can be provided by a CBSA approach developed to address the research problem \( RP \):

**RQ3:** “What software qualities can be satisfied and what guarantees are provided with a CBSA approach that deals with STO-reqs, disturbances and goal changes?”

1.2 Scientific Approach

To address the research problem \( RP \) and to obtain a solid research base, we combined three established research methods: systematic literature review, analytical method, and exploratory case studies. These methods are explained in details in Chapter 2 together with a comprehensive scheme of conducted research; in this
Section, we connect each of the methods to the research questions and give a brief generic overview of achieved results.

First, in order to obtain a systematic understanding of current research efforts in the application of control theory to adapt software systems and to find approaches that can deal with STO-reqs (RQ1), we performed a systematic literature review (SLR) following the guidelines from [96]. A systematic literature review is an established method to collect and analyze data within a certain topic of interest. In the conducted SLR, we investigated the trends of CBSA research (e.g., motivation to use control theory in software adaptation, research focus, application domains, etc.), model paradigms and adaptation solutions used in CBSA research, and types of goals and guarantees achieved with CBSA approaches. During the SLR we found an approach that became a leverage for a part of research presented in this Thesis.

After the SLR, we applied the analytical method, a research method used to develop, analyze and validate a formal theory or set of axioms [17]. As a first step of the analytical method, we designed a CBSA approach required to answer RQ2, which includes a software model and a control solution. Then, we analyzed the behavior of software systems designed with this CBSA approach through the application of mathematical techniques and formally verified a number of guarantees. This provided an answer to a part of RQ3 regarding the guarantees.

Finally, to address the part of RQ3 regarding software qualities, we applied informal exploratory case studies to the developed CBSA approach. An exploratory case study method offers the means to analyze a particular phenomenon, obtain new insights and ideas for research by studying one or more case instances in which the phenomena is involved [132]. In our work we use the prefix “informal” as we did not use the real-life study context as suggested in the literature [161, 132]. Instead, we addressed RQ3 by performing case studies with two simulated software systems: an unmanned underwater vehicle (UUV) system (based on [134]) performing surveillance mission and a service-based medical assistance system (TAS exemplar [153]). During the case studies, we analyzed software qualities that are satisfied by the developed CBSA approach and experimentally verified the obtained guarantees.

1.3 Scope of Research

This section specifies the scope of conducted research. It is divided into two subsections: assumptions regarding the software system being adapted, and types of quality properties and guarantees being considered.

Assumptions and Scope of Applicability

In this research, we target a family of software systems that work under a number of assumptions. In particular, we assume that the software system being adapted:

- Is available and is equipped with basic infrastructure for consistent adaptation (support for monitoring, adding/removing requirements, etc.).
1.3 Scope of Research

- Has multiple possibly conflicting requirements (STO-reqs) that are strict, i.e., a violation of requirement may lead to unwanted consequences. The requirements may change at runtime.

- Is a cooperative system in which entities have shared goals. Out of scope are real-time and competitive systems (entities that pursue their own goals). These systems require dedicated solutions.

- Has a limited, but potentially very high number of possible configurations (adaptation options) that can be selected according to the adaptation goals. The number of configurations may dynamically change over time.

- Performs communications and executes adaptations significantly faster than the pace of dynamics in the environment.

- Is not undergoing drastic changes in its behavior at runtime. For example, new components should not appear during system operation.

- Has to cope with the following types of uncertainties: disturbances (changes in the environment, inaccurate measurements and component failures) and goal changes (addition or removal of system requirements, or change of requirement values at runtime).

While these assumptions put restrictions on the target application domains, they hold for a large family of modern software systems. A typical example of such system is an unmanned underwater vehicle (UUV) used for oceanic surveillance. This system is equipped with a number of on-board sensors measuring a certain characteristic of the ocean, such as water pollution. These sensors have different characteristics: energy consumption, accuracy of measurements and the vehicle speed required for performing measurements. The UUV system works according to the list of assumptions presented above, namely:

- The UUV system is equipped with sensors for monitoring consumed energy, accuracy of measurements and vehicle speed. The system supports the change of sensor configuration at runtime.

- The adaptation goal of the UUV system is to select on-board sensors in such a way that the UUV covers as much distance as possible (T-req) with maximum measurement accuracy (O-req) using a specific pool of energy (S-req). These STO-reqs are strict as the adaptation should not lead to a loss of an expensive UUV equipment.

- The system entities (UUV sensors) share the same adaptation goal and work in cooperation to achieve this goal.

- The number of adaptation options is very high even with five on-board sensors because any combination of sensors can be activated at any time. In this case time is the factor that drastically increases the solution space, as using a sensor combination for 30 seconds will lead to a completely different outcome than the same combination used for 60 seconds.

- The underwater dynamics (e.g., current) are changing at much smaller rate than the pace of adaptation actions.
• The UUV architecture does not change during a mission and the system is prohibited to drastically change behavior at runtime. In critical scenarios the mission can be aborted in order to return the UUV to a safe location.
• The UUV has to operate in highly disturbed environment (changing current, pressure, etc.) and should be able to deal with different requirement change scenarios, such as an energy leak that requires minimizing energy consumption in order to safely deliver the UUV back to base.

Quality Properties and Control-Theoretical Guarantees

By specifying the STO-reqs, a stakeholder expects particular quality properties from the software. For example, satisfaction of a response time requirement should lead to a certain level of performance property\(^1\). In this work, we focus on adaption for a typical set of software quality properties\(^{[150]}\):

• Performance, the ability of the software to achieve a desired value for goals like throughput and response time.
• Efficiency, the extent to which the software uses the appropriate resources under stated conditions and in a specific context of use.
• Reliability, the capability of software to maintain its level of performance under stated conditions for a period of time.
• Accuracy, the degree of closeness of a measured value of a certain software property to that property’s true value.
• Business value, the additional value gained by software, e.g. increased profit, customer satisfaction, market share, revenue growth, etc.

We measured these quality properties in two software applications using different variables, see Table 1.1. Quality properties that cannot be linked to STO-reqs are out of the scope of this Thesis. An example of such property may be security.

| Table 1.1: Relation between quality properties and measured variables |
|-----------------------------|-----------------------------|-----------------------------|-----------------------------|-----------------------------|-----------------------------|
|                             | Performance | Efficiency | Reliability | Accuracy | Business value |
| UUV system                  | Scanned     | Energy     | Measur.     |          |               |
|                            | distance    | consumption| accuracy    |          |               |
| TAS exemplar               | Response    | Failure    | Service     |          |               |
|                            | time        | rate       | cost        |          |               |

As for the guarantees, we consider a standard set of guarantees analyzed in control theoretical literature \([79, 67, 108, 14]\). This set includes stability, overshooting, steady-state error, settling time, robustness to disturbances and inaccurate measurements (see Section 3.5 for detailed description of these guarantees).

\(^1\)Not to confuse properties with requirements. While requirements state what should be achieved by the software, the properties show what is achieved as a result of software operation.
1.4 Contributions

In accordance to the research problem \( RP \), this Thesis introduces two major contributions:

- A systematic survey of the CBSA research area that provides a comprehensive and structured view on the use of control theory to design self-adaptive software systems [1]. The survey answers \( RQ1 \) by describing trends of research on control-theoretical adaptation of software at the application and middleware level, model paradigms and adaptation solutions used in CBSA, and types of goals and guarantees achieved with CBSA.

- SimCA*: a reusable control-based engineering approach that allows to build self-adaptive software systems that satisfy STO-reqs under disturbances [137, 136, 2]. The approach can also deal with adjustment/activation/deactivation of requirements at runtime. SimCA* provides answer to \( RQ2 \) as it includes a formal model of software systems and an adaptation solution combining controllers with the simplex algorithm to handle STO-reqs, deal with disturbances and goal changes. The formal and experimental evaluation of SimCA* also answers \( RQ3 \). The formal evaluation includes verification of the following guarantees: stability, absence of overshoot, low settling time, robustness to disturbances, zero steady-state error, tracking of infeasible and unbounded solution. During the experimental validation we apply SimCA* to systems with strict requirements, the UUV system and the TAS exemplar, to measure the achieved software qualities (performance, reliability, etc.) and to confirm that the guarantees hold in those systems.

1.5 Thesis Overview

The remainder of this dissertation is structured as follows:

- Chapter 2 describes the scientific approaches that were used in our research. In particular, in this chapter we discuss systematic literature review, the analytical method, and informal exploratory case studies. We motivate the selection and give an overview of each research method.

- The research background is presented in Chapter 3. First, we provide a brief background on self-adaptive systems. Then, we outline the principles of control theory and introduce key terms such as control-theoretical feedback loop, setpoint, model, controller, PID, etc. After that, we introduce the two key components of SimCA*: the Push-Button Methodology and the Simplex method. Finally, we give an overview of control-theoretical guarantees and their formal analysis.
• The results of a systematic literature review on control-based self-adaptive software systems are given in Chapter 4. First, the chapter analyzes trends of CBSA research, such as the motivation to use control theory in software adaptation, research focus, application domains, assessment and validation methods, etc. Then, Chapter 4 discusses the model paradigms (e.g., model types, time dependency, model updates at runtime) and adaptation solutions (e.g., types of controllers, sensors and actuators) used in CBSA. The discussion of types of goals and guarantees achieved with CBSA approaches concludes the chapter.

• In Chapter 5, we describe a reusable CBSA approach called SimCA that is able to address S-reqs and O-reqs and provides control theoretical guarantees. The chapter includes: (1) A formal model of a self-adaptive software system which automatically updates according to runtime variations; (2) A control solution able to adapt the system according to SO-reqs in the presence of environmental disturbances and inaccurate measurements; (3) An experimental analysis of qualities achieved by self-adaptive software equipped with SimCA; (4) Formal analysis and experimental verification of the following guarantees: stability, overshooting, settling time, robustness to disturbances, steady-state error, detection of infeasible solution.

• Chapter 6 compares SimCA with an architecture-based ActivFORMS approach in multiple scenarios using the TAS exemplar. The comparison mainly focuses on the experimental comparison of software qualities, such as performance, reliability and cost, and guarantees, such as stability, robustness, reachability, safety, deadlock freeness, etc., achieved by those approaches.

• Chapter 7 describes SimCA*, a new approach based on SimCA that addresses new type of requirements where a value is kept above/below a threshold (T-reqs) and can deal with requirement sets that change at runtime (i.e., requirements can be adjusted, activated, and deactivated on the fly). Similar to Chapter 5, this chapter includes a formal model and a control solution useful for software adaptation, as well as the analysis of qualities and guarantees achieved by systems equipped with SimCA*.

• Finally, Chapter 8 concludes the dissertation. In particular, this chapter lists research contributions, outlines directions for future work, and provides some closing reflections.
Chapter 2

Research Approach

In this Chapter we provide a detailed description of the methods used to conduct the research, see Figure 2.1. The choice of these methods was based on the research problem \( RP \), the research questions and the specific goals we set during the different stages of our research. In particular, we discuss systematic literature review in Section 2.1, analytical method in Section 2.2, and informal exploratory case studies in Section 2.3. We motivate the selection and give an overview of each research method\(^1\).

![Figure 2.1: The detailed scheme of conducted research](image)

2.1 Systematic Literature Review

In order to obtain a systematic understanding of current research efforts in the application of control theory to adapt software systems and to address \( RQ1 \), we performed a systematic literature review (SLR) following the guidelines from [96].

\(^1\)Note that throughout Chapter 2 we do not distinguish between SimCA and SimCA* as they use the same research methodologies. As SimCA* is the more advanced version of the approach, we use it to describe the methodologies.
A systematic literature review is an established method to collect and analyze data within a certain topic of interest. As evident from its name, an SLR is conducted following a rigorous procedure. Typically, an SLR is performed as a first step to investigate a particular research problem and systematize knowledge within the researchers topic of interest. An SLR can also help in opening directions for future work by finding gaps in the existing research [21]. A distinct feature of an SLR is that the research problem, the data gathered to address this problem, and the step-by-step procedure to be followed during the review must be clearly stated to allow evaluation and reproduction for other experts within the chosen topic [96].

An SLR comprises three main phases: planning, executing, and reporting, see Figure 2.2. In the planning phase, the relevant research problem to be addressed by the review is defined. Then, the research questions are determined based on the problem. The final step of planning synthesizes a protocol for the review. This protocol describes the sequence of steps that will be followed to conduct the review, including the selection criteria for the studies, the types of data to be collected and the analysis procedures to be used. In the execution phase, the studies are selected, the data are collected from these studies and analyzed according to the protocol. In the reporting phase, the SLR results are summarized and reported, typically in a scientific publication.

![Figure 2.2: The scheme of a systematic literature review](image)

By conducting the SLR, we obtained a broad overview of the CBSA research field, including the trends of CBSA research (e.g., motivation to use control theory in software adaptation, research focus, application domains, assessment and validation methods, etc.), model paradigms (e.g., model types and time dependency, model updates at runtime) and adaptation solutions (e.g., types of controllers, sensors and actuators) used in CBSA research, and types of goals and guarantees achieved with CBSA approaches. Although the SLR showed that there is no approach in the literature able to address the research problem RP or the STO-reqs of software (RQ1), we obtained a number of ideas that helped us to create such a solution. In particular, we have found one reusable approach, the Push-Button Methodology [65], that is able satisfy a single S-req of software systems in the presence of disturbances and inaccurate measurements, and provides adaptation guarantees. The Push-Button Methodology became one of the leverages for the solution to RP presented in this Thesis, see Figure 2.1.

The description of SLR with detailed results is presented in Chapter 4.
2.2 Analytical Method

We used the analytical method to answer research question RQ2 and a part of research question RQ3 regarding the guarantees, see Figure 2.1. The analytical method is a research method used to develop, analyze and validate a formal theory or set of axioms [17]; it is typically applied in the formal computer science research [158]. The analytical method consists of four steps, see Figure 2.3:

- Applied to CBSA, suggesting a formal theory typically means treating software as a mathematical object and devising its model. For example, a model of a service-based system may specify how the amount of tasks sent to a particular service affects the output response time.
- Formal methods are then used extensively to develop the theory, which in CBSA terms means adjusting the initial software model and specifying an adaptation strategy. A formal method is a mathematical approach used to rigorously specify the behavior of a software system and to verify properties of interest. Abstraction is one of the key concepts when applying formal methods. It means that system features or details that are not relevant for the given problem are not included in the formal specification. Abstraction allows focusing on the problem under study and reusing formal specification for systems from different application domains. Different formal methods are used nowadays in self-adaptive software systems [155, 115].
- The analytical method does not use experiments or other empirical techniques in obtaining results. It rather applies formal methods to analyze the software behavior and the adaptation strategy. For example, a system engineer may calculate how fast a service-based system will react to a change in the response time requirement.
- In the final step of the analytical method, the analysis results are compared with the observations obtained from the running software system.

![Figure 2.3: The analytical method](image)

In this Thesis, in the first step of the analytical method we leveraged upon the Push-Button Methodology [65], found during the systematic literature review, in order to specify a reusable software model. Based on this model, in the second step we applied formal methods to design the SimCA* approach that addresses STO-reqs, deals with disturbances and goal changes (RQ2). Then, we analyzed the behavior of software systems equipped with SimCA* through the application of mathematical techniques and formally verified a number of guarantees (RQ3). As a last step, we applied informal exploratory case studies to SimCA*. These case studies are explained in the following section.

SimCA*, its formal specification and analysis are described in details in Chapter 7.
2 Research Approach

2.3 Informal Exploratory Case Studies

We used informal exploratory case studies to answer a part of research question RQ3 regarding the software qualities and to experimentally verify the previously analyzed guarantees, see Figure 2.1. The goal of an exploratory case study is to explore a certain phenomenon in order to understand or explain it, and to get new insights [132]. For example, in a service-based system that involves outdoor wireless communications, it could be beneficial to explore how weather affects data loss. The results of such case study may lead to a change of services being used during rain. An exploratory case study typically consists of four steps, see Figure 2.4. First, the problem to be addressed by the case study is defined. Second, the cases are selected according to that problem. Third, the cases are implemented and, fourth, the achieved results are analyzed.

![Figure 2.4: The exploratory case study method](image)

According to [161], the case study method “investigates a contemporary phenomenon within its real-life context”. In our work we use the prefix “informal” as we performed case studies with simulated systems and did not strictly follow the formal procedure of a case study suggested in the literature [161, 132]. The main motivation behind that is the higher controllability and lower complexity of simulated systems. For example, while real-life measurements performed by medical assistance services are costly, one can simulate thousands of measurements for free. A full-scale case study in a real-life context is a part of our future work, see Section 8.2.

In this Thesis, we used informal exploratory case studies to understand and characterize the behavior of software systems equipped with SimCA*. We selected two examples of software systems with strict requirements from different domains as case studies: a simulated unmanned underwater vehicle (UUV) system performing surveillance missions and a service-based medical assistance system (TAS exemplar). Although we did not formally follow a case study selection procedure, those cases pose typical requirements (STO-reqs) and require a solution for the research problem RP. Therefore, these cases are representative enough to validate our research results. During the case studies, we analyzed software qualities that are satisfied by SimCA* and experimentally verified the guarantees provided by the approach (RQ3). We also highlighted some qualities (such as security) that may be challenging to satisfy with SimCA*.

Informal exploratory case studies with both the UUV system and the TAS exemplar are described in Chapters 5, Chapters 6 and 7.
Chapter 3

Research Background

The goal of this Chapter is to provide a research background. We start with introducing the principles of self-adaptation in Section 3.1 and control theory in Section 3.2. Then, we describe the Push-Button Methodology in Section 3.3 and the Simplex method in Section 3.4. Finally, we discuss formal guarantees in Section 3.5.

3.1 Self-adaptive Software Systems

In this section we introduce the concept of self-adaptation, describe the components of an architecture-based self-adaptive software and show an example of such system.

Nowadays, the continuous availability of many computing systems has become a critical requirement. Typical examples are the web, telecommunication, manufacturing, transportation, etc. Due to the longevity of these systems and the dynamic conditions in which they have to operate, change is unavoidable in most of these systems. Consequently, software engineers require approaches that reduce the costs and risks of adapting and evolving these systems without incurring downtime. Driven by this trend, self-adaptation has been widely recognized as an essential capability of many software systems [42, 49].

A classic approach to engineer self-adaptive software is architecture-based adaptation [72, 123]. In this approach, a software system maintains an explicit architectural model of itself and reasons about that model to adapt at runtime to achieve the adaptation goals (e.g., self-heal, self-optimize). Figure 3.1 shows the four primary elements of an architecture-based self-adaptive system that interact with each other:

- Environment: a part of external world where the self-adaptive system is situated. The environment cannot be controlled directly by the system engineer, but can be effected by the software system.

- Software system: a system that is being adapted. The concern of the software system is the domain of interest; it interacts with the environment, providing functionality to users.

- Adaptation Goals: the goals that the software system should achieve as a result of adaptation. These goals are typically concerned with providing a
certain level of software qualities required by the stakeholder. Different approaches are used in architecture-based adaptation to specify the adaptation goals, e.g., fuzzy constraints [15] or probabilistic temporal logics [32].

- Adaptation System: a system that adapts the software system in order to satisfy the adaptation goals. A prominent approach to design the adaptation system is by means of a MAPE-K feedback loop (Monitor-Analyse-Plan-Execute-Knowledge) [94]. The monitor component of this loop gathers data to maintain a model of the software system and the environment in the knowledge component. The analyze component uses this model to identify whether the adaptation goals are achieved or not. If not, the plan component puts together a plan with the adaptation actions that the execute component applies to the software system.

A simple example that illustrates the components of an architecture-based self-adaptation is a service-based system (SBS). Consider a SBS (software system) that consists of two external services performing the same tasks (e.g., measuring temperature), but having different response time characteristic ($R_1$ and $R_2$). A stakeholder specifies the required response time $R_{goal}$ that SBS should achieve over time. The adaptation goal is to determine the percentage of tasks that should be send to each of the two external services ($x_1$ and $x_2$) in order to meet the requirement $R_{goal}$. The response times of the external services are a part of the environment as they change independently from a system engineer. Finally, a MAPE-K implementation of an adaptation system will measure the output response time $O_R$ during system operation (monitor), compare it with $R_{goal}$ (analyze) and adjust the percentage of tasks $x_1$ and $x_2$ if needed (plan and execute).

Substantial progress has been made in understanding the foundations and engineering principles of architecture-based adaptation. However, researchers face a number of challenges that call for exploring new perspectives on engineering adaptive systems. One important problem is how to provide guarantees in adaptive systems, which is particularly challenging given the fact that adaptive systems have to be designed with partial knowledge and consequently, runtime mechanisms are required. Applying classic techniques for providing guarantees (e.g., testing,
3.2 Control theory and software systems

This section provides a brief background on control theory and introduces the control terminology used throughout this dissertation. The elements of a typical control-based feedback loop are explained with an example scenario.

For many decades control theory has been used to adapt different physical systems from aircrafts to industrial product lines. In 1997, with the introduction of “dynamic feedback” [53] it started being explored as a solution to adapt computing systems. Seven years later, the book by Hellerstein et al. [79] unveiled the high potential of control theory in adaptation of computing systems. Guided by the need for adaptation guarantees, many of the follow-up efforts investigated the interplay between control theory and computing systems in general or control theory and software systems in particular. As a result of all these efforts, today control theory is considered a solid approach to design self-adaptive software which provides formal guarantees on the adaptation results [24, 48].

A typical control-based self-adaptive software system is a single-input, single-output system that consists of a feedback loop with the following elements (Figure 3.2, from left to right):

- **Goal**: a stakeholder requirement expressed as a particular value to be achieved by the system.
- **Control error**: the difference between the goal and the measured output.
- **Controller**: a component that, based on the system model and control error, computes the control signal required to achieve the goal.
- **Control signal**: a signal that adapts the computing system. The control signal triggers adaptation by effecting certain system knobs called actuators.
- **Disturbance**: any internal and external disturbance acting on the system.
- **Software system**: a system that is being adapted.
- **Measured output**: a measurable parameter of the computing system. The goal of adaptation is to keep the measured output as close as possible to the goal value. The mechanism measuring the output is called sensor.

![Figure 3.2: Control-based self-adaptive software system](image)
Using the same SBS example from Section 3.1, we illustrate the elements of a control-based feedback loop. In this case, the system goal is the required response time $R_{goal}$ that the SBS software system should achieve over time by sending tasks to the two external services. During operation, the system tracks the measured output response time $O_R$. Then, $O_R$ is subtracted from the goal $R_{goal}$ in order to receive the control error $e$. Based on $e$ and the system model (describe in the following paragraph), the controller calculates the control signal $x$ that includes the percentage of tasks that should be send to each of the two external services. The runtime variations in response times of the external services are considered disturbances.

A crucial element of a control-theoretic feedback loop that is not directly visible on Figure 3.2 is a model of the software system. In control theory, a model is represented as a system of difference or differential equations that describes the behavior of a software system [79, 14]. A controller is represented as a separate equation (or system of equations) designed based on this model to adapt the system. Controllers are designed for the following objectives: regulatory function (converge the system output to a particular value), optimization (minimize or maximize the output), disturbance rejection, or a combination of these objectives [79]. Depending on the problem at hand, the properties of a software system being adapted and the required controller objectives, different controller types can be used. Industrial practice mostly utilizes the Proportional-Integral-Derivative (PID) controller [157]. PID calculates the control signal to eliminate the control error [13]. The control signal is based on three terms: a proportional term that takes into account only the current value of control error, an integral term that takes into account past values of control error, and a derivative term that works with predicted future values of the control error. Optimal controllers, on the other hand, calculate a control signal that minimizes a cost function subject to certain constrains, e.g. maximize performance using a pool of limited resources. A particular representative of optimal controllers used to adapt computing systems is a Model Predictive Controller (MPC). Other types of controllers are less common. Any controller can be adaptive meaning that the controller equation or variables of this equation are updated during system operation. Controllers can also be composed into controller schemes, e.g. to solve a task in parallel or in cooperation, becoming much more complex than the one depicted on Figure 3.2.

In this Thesis, we use multiple adaptive PI-controllers $C_{pbm}$ (the derivative term is excluded meaning that the predicted error value is not taken into account) combined with the Simplex block in a hierarchical structure. The controller $C_{pbm}$ is described in Section 3.3, Simplex is explained in Section 3.4. As the system model and controller are both equation-based, it becomes possible to mathematically analyze and verify certain system properties; this is discussed in Section 3.5.
3.3 Push-Button Methodology

In this section we describe the Push-Button Methodology (PBM) [65] which serves as a basic component of SimCA/SimCA*. The use of PBM is illustrated on a scenario with a service-based system.

Historically, many of the CBSA approaches were developed to address specific problems [3, 5]. Therefore, new problems required either modifying an existing solution or developing an entirely new approach. In 2014, the authors of [65] introduced a reusable PBM methodology that automatically creates a software model and a controller that adapts software to meet a single S-req specified by the stakeholder. As such, CBSA solutions for different problems can be developed with PBM by slightly tuning the controller parameters and without control expertise.

PBM requires the following input: a working software system, an S-req to be achieved and the tools to measure the system output related to this S-req, and a tunable parameter that can change the behavior of software in order to satisfy the S-req. Having this input, PBM automatically synthesizes a system model and creates a controller in two phases, see Figure 3.3:

![Figure 3.3: The Push-Button Methodology](image)

In the *model building* phase the following linear model of the software is automatically constructed:

$$y(k) = \alpha \times \eta(k-1)$$  \hspace{1cm} (M_{pbm})

Where $y$ is the measured output for the S-req property, $\eta$ is the value of tunable parameter that can change the behavior of software, $\alpha$ is a model coefficient and $k$ is a discrete time instance. The coefficient $\alpha$ reflects how different values of $\eta$ influence $y$; $\alpha$ is calculated at runtime by feeding the systematically sampled values of the actuator $\eta$ into the software and measuring the resulting output $y$.

In the *controlling* phase, the following PI-controller that works on the model $M_{pbm}$ and adapts the software is automatically created:

$$\eta(k + 1) = \eta(k) + \frac{1 - p}{\alpha} \times e(k + 1)$$  \hspace{1cm} (C_{pbm})

During each adaptation step, this controller selects the tunable parameter that changes the behavior of software $\eta(k + 1)$ based on its value at the previous adaptation step $\eta(k)$, model coefficient $\alpha$ calculated during model building, system parameter called pole $p$, and on the error $e(k + 1)$ between the target and the measured value of an S-req property. The pole $p$ is a value that can be changed by a
system engineer in the interval (0,1); it is used for controller tuning and allows to trade-off the guarantees obtained. In general, PBM provides the following set of guarantees: system stability, absence of overshoot, settling time and robustness. These guarantees are discussed in details in Section 3.5.

To cope with small disturbances and system dynamics, PBM updates the value of $\alpha$ during system operation using a Kalman filter [147], which makes controller $C_{pbm}$ adaptive. When a drastic change in the system behavior occurs (e.g., a component failure), PBM triggers a complete model rebuilding phase, followed by a new controlling phase. These solutions allow the reuse of a simple linear model $M_{pbm}$ for different types of software systems.

In order to illustrate how PBM works, we again use the SBS scenario from Section 3.1. Here, a user specifies SBS as a software, the amount of tasks sent to each of the two external services as a tunable parameter and the response time as an S-req. Having the required input, PBM automatically creates the model $M_{pbm}$, where $y$ is the measured response time $O_R$ and $\eta$ is the amount of tasks that should be send to the first external service $x_1$, while $\alpha$ approximates how different numbers of tasks affect the system response time. Based on this model, PMB creates a controller $C_{pbm}$ that guarantees the required response time by changing the amount of tasks sent to each of the services at runtime. Namely, it selects the number of tasks to be send to the first external service $x_1$ based on the previously sent number of tasks $x_1(k-1)$ and the difference between required response time $R_{goal}$ and measured response time $O_R$ multiplied by $\frac{1-p}{\alpha}$ (recall that $\alpha$ is calculated automatically during model building, while $p$ is chosen by the system engineer).

### 3.4 Simplex

In this section we explain the simplex method which is a second basic component of SimCA/SimCA*. In mathematics, the simplex method is used to solve a problem of optimizing (minimizing or maximizing) a certain objective function subject to a number of constraints. This problem is commonly known as a linear problem written in the standard form:

$$
\min\{c^T x \mid Ax \leq b; x \geq 0\} \quad (3.1)
$$

where $x$ represents the vector of variables (to be determined), $c$ and $b$ are vectors of (known) coefficients, $A$ is a (known) matrix of coefficients, and $(\cdot)^T$ is the matrix transpose\(^1\) [46].

To simplify understanding of the problem, we rewrite (3.1) as a system of equations using a modified SBS scenario from Section 3.1. In this scenario, SBS still consists of two external services performing the same task, but now those services have different invocation cost ($I_1$ and $I_2$) and failure rate ($F_1$ and $F_2$) characteristics in addition to response times ($R_1$ and $R_2$). The system goal is now to calculate the amount of tasks send to the external services $x_1$ and $x_2$ in a way that minimizes

\(^1\)Matrix transpose is an operator that turns all rows of a given matrix $A$ into columns and all columns of $A$ intro rows. The resulting matrix is denoted $A^T$.\]
the output failure rate, spends the available budget ($I_{goal}$) and keeps the response time below a certain value ($R_{goal}$). In this case, 3.1 will look as follows:

Minimize Failure rate:

$$\min \left[ F_1 \times x_1 + F_2 \times x_2 \right]$$

Subject to:

$$\begin{align*}
I_1 \times x_1 + I_2 \times x_2 &= I_{goal} \\
R_1 \times x_1 + R_2 \times x_2 &\leq R_{goal}
\end{align*}$$

(3.2)

A relatively easy way to explain how simplex solves the linear problem (3.1 and 3.2) is by means of a geometric representation. Geometrically, the system of equations (3.1) can be represented as a three-dimensional figure called a convex polyhedron (Fig. 3.4). Finding a solution of a problem with simplex starts with finding a first extreme point of the polyhedron known as a basic feasible solution. In case there is no basic solution, the system is considered unsolvable. When the extreme point is found, the simplex method starts moving along the polyhedron boundaries from one extreme point to another following such a path that the value of the objective function $c^T x$ becomes lower at every step [46, pp.63-98]. In other words, the simplex method does not look through the entire solutions space, but takes a specific path between the extreme points of the solution. The optimal solution (vector $x$) is reached when the value of the objective function $c^T x$ reaches its minimum, i.e. when the simplex method cannot find a new extreme point that decreases the objective function.

Figure 3.4: Solving tasks with simplex: a geometric representation

3.5 Formal Guarantees

In general, the purpose of guarantees is to assure that the target software system satisfies stakeholders requirements independently of changing internal/external conditions and in a most efficient way. In this section we explain the standard control theoretical guarantees and demonstrate them with an example scenario.
Types of Control-Theoretical Guarantees

In order to explain the guarantees, we first define the concept of transient and steady state. A software in a steady state performs its usual operations, e.g., a service analyzes the outdoor temperature. If, due to an unexpected failure, the service stops working, a controller triggers a restart. At that point the service switches to a transient state. When the service starts analyzing the temperature after the restart, it is again considered to be in a steady state. In other words, a transient state shows a period when controller tries to stabilize the system and return it to normal operation.

According to [79], control theory can guarantee four main system properties, see Figure 3.5:

- Stability: ability of the system to converge to a value (setpoint).
- Steady-state error: having a small steady-state error means keeping the measured output close to the setpoint in a steady state.
- Settling time: the time needed for a controller to set the measured output close to the steady-state value.
- Overshoot: the amount by which the measured output surpasses the setpoint during transient state.

Additionally, a number of studies in control theory discuss the guarantee of robustness. Robustness indicates the amount of perturbation the system can withstand without moving to an unstable state, where the system output does not converge to a setpoint.

To understand the importance of these guarantees for a software system we look at the SBS scenario from Section 3.1. Generally speaking, stability may affect all software qualities that are subject of adaptation. In the SBS case, the system instability, i.e., the inability to converge to the response time goal $R_{goal}$, would lead to deviations in the output response time $O_R$ causing unstable performance. The same can be concluded about high steady-state error, as the constantly oscillating value of $O_R$ is not something that a stakeholder expects from SBS. The settling time guarantee may affect different software qualities, but in general a fast return to a steady state wastes less time and resources in a transient state. In the SBS scenario, smaller settling time will result in an average output response time $O_R$ that is closer to the goal $R_{goal}$. Avoiding overshoot avoids penalties on the respective software quality, so an overshoot of system response time in SBS would
violates the $R_{\text{goal}}$. Finally, robustness directly influences system reliability. If SBS is expected to work in a highly disturbed environment, where response times of external services $R_1$ and $R_2$ drastically change all the time or where the service communication channel is influenced by noise of high amplitude, then low robustness would simply lead to instability, again causing undesirable performance.

**Achieving Guarantees**

The analysis of guarantees performed throughout this dissertation is based on mathematical analysis technique called Z-transform [79]. Z-transform allows to convert a discrete time control signal (which is basically a list of values) into a frequency domain representation (a sum of values) and vice versa. Z-transform of a self-adaptive system $G(z)$ shows how the system output $O(z)$ is related to the input $S(z)$: 

$$G(z) = O(z)/S(z).$$

$G(z)$ is called the transfer function. Mind that the transfer function may be freely converted to its discrete time representation and back to Z-representation. A particular parameter of interest in the transfer function is the pole $p$. By definition, when $z \to p$, $G(z) \to \infty$. The pole may be tunable or set to a specific value depending on the controller design.

Mathematical analysis of the transfer function $G(z)$ and its discrete time counterpart provides steady-state error and robustness guarantees. Stability and absence of overshoot are achieved by keeping the pole $p$ in a certain interval; settling time is calculated based on the pole value as well.

We now illustrate analysis of guarantees with an example. Consider the following transfer function in the discrete time domain: 

$$O(k) = S \times (1 - p^k),$$

where $k$ is a discrete time instance. This is a real transfer function used in PBM and SimCA. It’s Z-transform is: 

$$G(z) = (1 - p)/(z - p).$$

By analyzing the discrete time equation, it can be concluded that the system with such transfer function will be stable whenever pole $p$ belongs to the open interval $(0, 1)$. Proof: whenever $p > 1$, $p^k$ is unbounded making $O(k)$ decreasing exponentially; whenever $p < 0$, $p^k$ is positive if $k$ is even and is negative if $k$ is odd, making $O(k)$ oscillating.

To evaluate the system output during steady-state, we again look at the example transfer function in the discrete time. By definition, a steady state means $k \to \infty$. As $p \in (0, 1)$, $p^k \to 0$ in a steady state. Then, the system output during steady state is $O(k \to \infty) = S \times (1 - p^k) = S$. Hence, the steady-state error equals: 

$$\Delta e = S - O(k \to \infty) = 0.$$ 

The analysis of remaining guarantees is described in details in Chapter 5.
Chapter 4

A Systematic Literature Review on Control-Based Software Adaptation

In this Chapter, we report the outcomes of a Systematic Literature Review (SLR) that investigated the research efforts in Control-Based Software Adaptation (CBSA) research field. In total, we extracted data from 42 primary studies selected from 1512 papers that resulted from an automatic search. The studies were gathered from 41 venues in control theory, software/systems engineering, and adaptive systems published from January 2000 to June 2016.

The main goals of the SLR were to provide a general overview of a CBSA research field and to answer the research question RQ1 “What CBSA approaches are used to deal with STO-reqs in the existing literature?”

The SLR identified:

• Trends of research on control-theoretical adaptation of software at the application and middleware level;

• Model paradigms and adaptation solutions used in CBSA;

• Type of goals achieved with CBSA approaches presented in the literature and types of guarantees provided.

Reflecting upon results of the SLR, we did not find a CBSA approach that could effectively address the research problem RP or satisfy STO-reqs with guarantees. However, we have found the Push-Button Methodology (PBM), an approach that satisfies a single S-req with guarantees. PBM became a leverage for our own approach that deals with RP, see Chapter 5.

This Chapter is a copy of our journal article published in Transactions on Software Engineering [1]. The review supporting material can be found in [135].
Control-Theoretical Software Adaptation: A Systematic Literature Review

Abstract

Modern software applications are subject to uncertain operating conditions, such as dynamics in the availability of services and variations of system goals. Consequently, runtime changes cannot be ignored, but often cannot be predicted at design time. Control theory has been identified as a principled way of addressing runtime changes and it has been applied successfully to modify the structure and behavior of software applications. Most of the times, however, the adaptation targeted the resources that the software has available for execution (CPU, storage, etc.) more than the software application itself. This paper investigates the research efforts that have been conducted to make software adaptable by modifying the software rather than the resource allocated to its execution. This paper aims to identify: the focus of research on control-theoretical software adaptation; how software is modeled and what control mechanisms are used to adapt software; what software qualities and controller guarantees are considered. To that end, we performed a systematic literature review in which we extracted data from 42 primary studies selected from 1512 papers that resulted from an automatic search. The results of our investigation show that even though the behavior of software is considered non-linear, research efforts use linear models to represent it, with some success. Also, the control strategies that are most often considered are classic control, mostly in the form of Proportional and Integral controllers, and Model Predictive Control. The paper also discusses sensing and actuating strategies that are prominent for software adaptation and the (often neglected) proof of formal properties. Finally, we distill open challenges for control-theoretical software adaptation.

4.1 Introduction

Software applications are, more than ever, forced to deal with change [124, 94]. The need for continuous availability of software is forcing developers to consider change as part of the development process. Software should be able to execute in conditions that differ from the ones it was initially designed for, for example because new hardware is available with respect to what was envisioned at design time [85]. Moreover, software should execute with incomplete knowledge of the execution environment and conditions and face changing requirements during operation [139]. Consequently, software engineers are developing new techniques to handle change at runtime without incurring into penalties and downtime, giving birth to what is commonly referred to as software self-adaptation [42, 49].
Different alternative approaches have been proposed for the design of self-adaptive software, a prominent one being architecture-based adaptation [123, 71, 102, 156]. In the architecture-based approach, the software generates and updates an explicit architectural model of itself and uses it to reason about adaptation. Applying classic techniques like testing and model checking for providing assurances at runtime is challenging, especially because these techniques assume the availability of accurate models of the software behavior. The partial knowledge available at design time represents a challenge for architecture-based approaches, in particular regarding the formal guarantees that can be provided [30, 151].

Self-adaptive software must deal with change at runtime, when the knowledge of how to handle this change becomes available. The software engineer includes mechanisms to handle runtime variations in the software design and implementation [133]. Most of these mechanisms use feedback from the software and the environment to adapt some part of the execution and ensure that the requirements are met under changing execution conditions. Control theory was identified as a discipline that could offer insight on the design of adaptation mechanisms with formal guarantees [24, 79, 48, 166].

So far, most research on control-theoretical adaptation of computing systems focused on controlling lower-level elements/resources of the technology stack (CPU, storage, bandwidth, etc.) [4, 51, 167]. With respect to the adaptation of resource allocation, applying control theory to adapt the software behavior is a more complex problem [65, 7, 22], due to the difficulty of accurately modeling software, to the types of requirements and their tradeoffs [8] and to the need of instrumenting software to obtain sensor measurements and actuators [26, 82].

Research efforts applying control-theoretical adaptation to software exist [58, 20, 65, 137]. However, the results of these efforts are scattered and consequently, there is no clear view on state of the art. This calls for a consolidation of the knowledge on the application of control-theoretical principles to software adaptation. Such knowledge would provide understanding of the basic engineering principles, including the software models and the control mechanisms, as well as the types of achieved goals and provided guarantees.

To systematize the mentioned knowledge, we performed a systematic literature review, following a well-defined methodology that identify, evaluate and interpret the relevant studies with respect to specific research questions and topics of interest [96]. In the review, we have analyzed research results from 41 main conferences and journals in software/systems engineering, adaptive systems and control theory, in the period 2000-2016. The focus of the study is on three different aspects: models, control strategies and formal guarantees.

More precisely, in software engineering, models typically rely on architectural concepts, like components and connectors. In control theory, on the contrary, models are typically behavioral based – in the case of discrete event control – and equation-based – for discrete and continuous time control. One of the crucial topics of this survey is the role of models in control-theoretical software adaptation. The second topic this survey focuses on is control structures. In control theory, a controller structure is chosen based on the characteristics of the specific problem, like the presence or absence of model uncertainties or the required speed of...
convergence towards goals. Finally, in software engineering, development time techniques such as code reviews and model checking are usually coupled with runtime techniques like quantitative verification [32] to provide guarantees on the adaptation process. In control theory goals are usually expressed as setpoints and guarantees are expressed and obtained at design time, in terms of the ability to reach the desired objective whenever feasible. Guarantees are typically given on the model, and their validity is evaluated against model inaccuracies and parametric uncertainty.

The remainder of this paper is organized as follows: Section 4.2 provides information about the specific focus of the review, Section 4.4 provides some background on control theory, Section 4.5 contains information about related surveys and efforts, Section 4.6 discusses the research methodology used for this survey, Section 4.7 describes the findings of this survey. From the analysis, we have derived some insights that helped us to outline relevant challenges for future research, that are described in Section 6.5. Finally, Section 4.9 discusses threads to validity, Section 4.10 concludes the paper.

4.2 Focus of the Literature Study

This section describes the focus of the conducted literature review in detail. We distinguish between the software system being adapted, discussed in Section 4.2, and the control technique being applied, described in Section 4.2.

Software Adaptation

Control-theoretical adaptation was used in a variety of computing systems [79, 126] with different objectives. This systematic literature review focuses on software adaptation. Software adaptation here refers to the actual adaptation of a running software application and to the adaptation throughout the software development live cycle, from requirements to design, construction, testing, deployment, software maintenance, and evolution. Figure 4.1 shows the typical structure of modern computing system in three layers. Each layer is illustrated with the example elements from three domains: webservice, warehouse logistics, and cloud applications.

The bottom system layer includes resource elements such as CPU, storage, sensors and cloud hardware resources. The lower part of the middleware layer incorporates software that can be mapped directly to physical and virtual elements in the system layer. Examples of elements in this sub-layer are application servers, software drivers, and the platform services. Adaptation at the system and lower middleware level has been reviewed in the past [126], mostly providing a view on resource provisioning techniques based on control-theory. In these problems, resources are generally treated as flows and the control problem is often mapped to flow regulation [4, 25].

Adaptation refers to actions that lead to change of the software application, from architecture reconfiguration to component replacement, to switches in the application mode, to parameter changes.
4.2 Focus of the Literature Study

<table>
<thead>
<tr>
<th>Application</th>
<th>Example 1</th>
<th>Example 2</th>
<th>Example 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain-specific software element</td>
<td>Webservice application</td>
<td>Warehouse transportation application</td>
<td>Cloud application</td>
</tr>
<tr>
<td>High-level reusable services</td>
<td>Workflow engine</td>
<td>Communication service, Planning software</td>
<td>Software services</td>
</tr>
<tr>
<td>Services mapped to resources</td>
<td>Application servers</td>
<td>Gateways, Software drivers</td>
<td>Platform services</td>
</tr>
<tr>
<td>Physical and virtual elements</td>
<td>CPU, network, storage, etc.</td>
<td>Robots, sensors, etc.</td>
<td>Infrastructure, hardware resources</td>
</tr>
</tbody>
</table>

**Figure 4.1:** The three layers representing modern computing systems. The focus of the review is displayed in grey.

Adaptation of the higher middleware layer and the application layer is fundamentally different from adaptation performed at lower levels. Differences include:

- Software exhibits three possible adaptation dimensions: requirements, structure, and behavior [8];
- Context, goals and requirements are domain-dependent and can change during runtime [133, 139];
- There is a necessity to use complex and potentially multiple system models simultaneously [65, 26];
- The choice of proper sensors and actuators for adapting software can be challenging [48];
- The design space for the adaptation of software applications is often multidimensional [7, 22];
- Usually, there is a complex interplay between the qualities that are subject to adaptation on the one hand and the space of available adaptation options on the other hand [66].

This review includes studies where control was applied to software elements, rather than to hardware resources or low-level elements in the technology stack. The grey area in Figure 4.1 depicts the focus of this study. The focus avoids cluttered results that mix models, controllers, goals and guarantees for different layers of computing systems.

**Control Techniques**

The focus of this study is also restricted by the control techniques used for the adaptation system design. Figure 4.2 shows an extension of the taxonomy of potential control techniques proposed in [115, 48].
In this literature review, we analyze studies that use either classic or advanced control theory to adapt software systems. This includes the use of Proportional Integral Derivative (PID) controllers and controllers synthesized with pole placement or loop shaping techniques, Model Predictive Control (MPC) regulators, optimal controllers like Linear Quadratic Regulators (LQG), $H_{\infty}$ controllers. We also delve into adaptive and stochastic control.

Despite the fact that knowledge-based control approaches are usually considered closely related to control-theoretical ones, the foundations of the two techniques are quite different. Knowledge-based strategies rely on building an ontology that is then used to decide what is the best strategy to achieve a specific goal [86, 87], while purely control-theoretical approaches rely on models. In the case of knowledge-based strategies – including fuzzy controllers, rule-based control, case-based reasoning, heuristics, and machine learning – the controller cannot rely on cases that it has not seen during its training phase. Equation-based models are in nature approximations of reality and the use of a such models comes with the underlying assumption the behavior of the system in a point in between observations can be interpolated based on the data available.

Logical control based on discrete event systems (DES) is a substantial part of control theory. However, in contrast to classic control, DES approaches rely on transition system models, such as Petri nets and timed automata [146, 77, 121], for which the software engineering community has developed a lot of formal methods and tools that allow to model and assess the behavior of the system as a whole (e.g., particular logics and model checking). In this literature review we focus on adaptation of software based on models and guarantees that classical control theory can offer. Hence, we excluded studies that apply DES to realize adaptation. Finally, queuing theory and game theory have offered, in recent years, a basis for the development of different adaptation mechanisms [75, 113]. However, these mechanisms are quite different compared to the basic control-theoretical approaches and the type of guarantees that can be provided are different, making it difficult to compare approaches. Notably, studies that combine the use of queuing models with classic or advanced control strategies are still in the focus of our review.

Due to their different nature with respect to control-theoretic adaptation, we exclude knowledge-based, discrete event, game-theoretic and queuing-based approaches from this literature study.
4.3 Self-adaptive Systems Background

This section provides a brief background on self-adaptive systems based on [148]. We start with explaining the principles and motivation of self-adaption. Then we introduce basic concepts and illustrate the realization of the adaptation-specific elements with typical examples from a software engineering perspective. For additional reading, we refer the interested reader to [42, 142, 19, 54, 83, 133].

Principles of Self-adaptation

The term self-adaptation is not precisely defined in the literature. [42] refers to a self-adaptive system as a system that “is able to adjust its behavior in response to their perception of the environment and the system itself” [24] adds that “the self prefix indicates that the system decides autonomously (i.e., without or with minimal interference) how to adapt or organize to accommodate changes in its context and environment.” These researchers take the stance of an external observer and look at a self-adaptive system as a one that can handle changing external conditions, resources, workloads, demands, and failures.

[71] contrasts traditional adaptation mechanisms, such as exceptions in programming languages and fault-tolerant protocols, with mechanisms that are realized by means of a feedback loop to achieve various goals by monitoring and adapting system behavior at runtime. [7] refer in this context to “disciplined split” as a basic principle of a self-adaptive system, referring to an explicit separation between a part of the system that deals with the domain concerns and a part that deals with the adaptation concerns. Domain concerns relate to the goals for which the system is built; adaptation concerns relate to the system itself, i.e., the way the system realizes its goals under changing conditions. These researchers take the stance of a system engineer and look at self-adaptation from the perspective of how the system is conceived.

From these two perspectives, [148] identifies two basic principles that complement one another and determine what is a self-adaptive system: (1) the external principle: a self-adaptive system is a system that can handle changes in its environment, the system itself and its goals autonomously (i.e., without or with minimal human interference), and (2) the internal principle: a self-adaptive system comprises two distinct parts: the first part interacts with the environment and is responsible for the domain concerns (concerns for which the system is built); the second part interacts with the first part and is responsible for the adaptation concerns (concerns about the domain concerns).

Conceptual Model of Self-adaptive Software

Figure 4.3 shows a conceptual model of a self-adaptive software system. It consists of four basic elements: environment, software system, adaptation goals, and adaptation system. These basic elements are abstract and very general, i.e. they do not depend on a type of deployment, coordination between system components and the decision-making entity. A wide variety of approaches have been studied and
applied that realize the basic elements in different ways. We illustrate the realization of the adaptation-specific elements (adaptation goals and adaptation system) with typical examples from a software engineering perspective.

Environment refers to the part of the external world with which the self-adaptive system interacts and in which the effects of the system will be observed and evaluated. The environment can include both physical and virtual entities. As the environment is not under control of the software engineer, there may be uncertainty in terms of what is being sensed or what will be the result of effecting actions. An example of the environment of a robotic system is the physical environment in which the robots can move, but also the drivers of the cameras that the robots use to sense its surrounding.

Software System comprises the application code that realizes the system goals for the domain at hand. To that end, the software system senses the environment and can effect the environment. For example, a robot can plan a path to perform a transportation task. During its mission, it can use a camera to detect obstacles, compute an alternative path if necessary, and steer the vehicle around obstacles to avoid collisions.

Adaptation Goals are goals of the adaptation system over the software system; they usually relate to qualities of the software system. [94] distinguishes between four types of high-level adaptation goals: self-configuration (i.e., systems that configure themselves automatically), self-optimization (systems that continually seek ways to improve their performance or cost), self-healing (systems that detect, diagnose, and repair problems resulting from bugs or failures), and self-protection (systems that defend themselves from malicious attacks or cascading failures). For example, a self-optimization goal of a robot may be to ensure that a particular number of tasks are achieved within a certain time window under changing operation conditions, e.g., dynamic task loads or reduced bandwidth for communication.

Adaptation goals are often expressed in terms of the uncertainty they have to deal with. Example approaches are the specification of quality of service goals using probabilistic temporal logics [32], and fuzzy goals whose satisfaction is represented through fuzzy constraints [15]. Adaptation goals are typically a first-class
entities at runtime, enabling the adaptation system (see below) to reason about the adaptation goals during operation.

Adaptation System manages the software system. To that end, the adaptation system comprises adaptation logic that deals with the adaption goals. To realize the adaptation goals, the adaptation system senses the environment and the software system and adapts the latter when necessary. For example, to achieve the required number of tasks within a certain time window under peak load, the robots give priority to particular types of tasks. Conceptually, the adaptation system may consist of multiple layers where the upper parts manage the underlying subsystems.

The adaptation logic can be realized with different approaches. A classic approach applied in software engineering is to model the adaptation logic in the form of four components, Monitor, Analyze, Plan, and Execute that share common Knowledge (often referred to as MAPE-K [94]). The Monitor acquires data from the managed element and the environment, and processes this data to update the content of the Knowledge element accordingly. The Analyze element uses the up-to-date knowledge to determine whether there is a need for adaptation of the managed element. To that end, the Analyze element uses representations of the adaptation goals that are available in the Knowledge element. If adaptation is required, the Plan element puts together a plan that consists of one or more adaptation actions. The adaptation plan is then executed by the Execute element that adapts the managed element accordingly.

A key aspect of self-adaptation is to provide guarantees for the compliance of the adaption goals of self-adaptive systems that operate under uncertainty. A pioneering approach that deals with this challenge is quantitative verification at runtime. [32] applies this approach in the context of managing the quality of service in service-based systems. Extensive research has shown that providing guarantees for the compliance of the adaption goals with traditional software engineering approaches (ranging from traditional testing and sanity checks to model checking) remains a challenging problem [151]. This is one of the key reasons why researchers started exploring alternative paradigms such as the application of control theory to realize self-adaptation.

4.4 Control Theory Background

This section introduces some background on control theory, and defines the terminology that will be used for the analysis of the studies. For further reading on control theory, the reader can refer to [108, 14, 106, 122, 110, 107].

Steady state and Transient Phase

In physical systems, when an input is applied to an object, this object usually reacts to the input. For example, if a person kicks a ball on a grass field, the force applied to the ball will make it move until a specific location. If one measures the position of the ball compared to the initial position, the signal will show a movement until the ball will stop (due to friction). The signal has clearly two distinct behavior. In a first phase (the transient phase), the ball will move, depending on the applied
force. In a second phase, in absence of other forces, the ball position will settle to one specific location. This second phase is called steady state. A system is in steady state when the initial force applied has vanished its effects and it is in the transient phase while the effect of the initial force can still be observed. In general, the output signal of a system in the steady state is not necessarily a constant. For some systems, for example, the output can be a cyclic behavior.

As a parallelism with programming, one may think about a system in steady state as a piece of software, always repeating the same operations. If something happens in the software, some other routines can be started, to handle the interrupt. When these handling routines terminate, the software can go back to the original state of repeating the same operations.

**Feedback and Feedforward Control**

Figure 4.4 shows the basic block diagram of a feedback control scheme, applied to a software system. From left to right, the *Setpoint* represents the goal that the adaptation needs to achieve – typically a non-functional requirement such as a specific response time or a reliability value. Based on the value of the desired goal and the corresponding *Measured Output* an error is computed that is used by a *Feedback Controller* to compute the *Control Signal*. The control signal adapts the *Software System* such that the output gets as close as possible to the Setpoint. The \( -1 \) block indicates that the value of the feedback signal is inverted, that is, the Error is computed as: \( \text{Setpoint} + (-\text{MeasuredOutput}) \). During normal operating conditions the system reaches a steady state. When the measured output changes due to external *Disturbances*, the system enters a transient phase, where the feedback controller applies an appropriate control signal to handle the disturbances to bring the system back to the steady state. Figure 4.5 shows the basic block diagram of a feedforward control scheme. A *Feedforward Controller* takes into account the *Setpoint* and the values of external *Disturbances*, and produces a *Control Signal* that compensates for the disturbances.

To grasp the difference between feedback and feedforward control, imagine a person driving to a predefined destination. Feedforward control is the act of checking a map beforehand and memorizing it, computing the best strategy to get to the destination and applying this strategy when driving. Feedback control is the act of checking a navigation device that provides the current position and distance from the destination. A model of the map is still needed to define the direction, but this model is used during the navigation to refine the current navigation strategy.
In general, control strategies are developed to counteract the effect of disturbances on systems. In the case of software systems, these disturbances may come from the environment or from the internals of the software itself. The underlying assumption for the application of control is the ability to measure the output of the software behavior that must be kept under control. A measure of the disturbances, on the contrary, can be beneficial for the setup of a feedforward strategy, but is not necessary.

While the main purpose of feedback control has historically been disturbance rejection, the coupling of the feedforward block and the feedback one has the purpose of following a setpoint. Setpoint tracking is the other objective of the application of control.

**Taxonomy of Classic Controllers**

In the blocks corresponding to feedback and feedforward controllers, one can implement different control strategies, ranging from classical control to more advanced techniques. Over the years, a lot of control techniques have been studied. The first group of techniques is generally called state-feedback controllers. These are controllers that use information about the state of the system to decide on a control signal [157]. One of the earliest strategies based on state feedback that has been developed is the bang bang controller, which consists in turning on or off a specific actuator, for example opening a valve to let water flow or closing it. In computing systems, this is usually the controller employed for admission control strategies, where requests are either admitted or rejected. Other state feedback controllers are regulators based on Pole Placement, Deadbeat Controllers and Proportional Integral and Derivative (PID) controllers. The PID controller is the most common controller and covers about 90% [157, 13] of the industrial applications of control. It is based on computing a control signal as a function of the error between the desired system behavior and the current system behavior.

The second group of techniques is called optimal control. In optimal control, the control value is obtained to minimize a cost function, possibly subject to some constraints. Typically, the objective is to maximize control performance, given prescribed guarantees [119, 165]. Whenever the cost function is a quadratic function, and the constraints contain linear first-order dynamic constraints, the problem can be classified as a Linear Quadratic (LQ) optimal control problem. A special case is the Linear Quadratic Regulator (LQR) [138].

A particularly successful heuristic for optimal control under constraints is Model Predictive Control (MPC) [112, 34]. MPC predicts the future behavior.
from the current system state under a particular control action and selects the input sequence that minimizes the chosen cost function. Only the first step of that input sequence is applied and at the next time step the new system state is determined and the process repeated, according to receding horizon principle.

Together with these control strategies, there is Robust Control [119], which is based on building a control strategy that makes the system behave in a specific way, despite variation of involved parameters. In general, robustness to model inaccuracy is a property of all control strategies, but there are design techniques to develop controllers that are specifically aimed at maximizing robustness.

**Composition of Controllers**

Controllers can be composed by combining multiple feedback and/or feedforward controllers that interact with each other. For example, the feedback controller block may correspond to one of the following: multiple cascaded controllers; a hierarchical structure where the control signal is determined by controllers coupled together; controllers working in parallel or concurrently. When controllers are composed, the feedforward control signal is incremental with respect to any other control signal computed in the system (for example, from a feedback controller block). If no other controller is present then the feedforward control signal is applied directly to the software system. The main goal of combining feedback and the feedforward controllers is that the latter can take care of the part of disturbances that can be modeled, while the former can deal with disturbances that are not known a priori. The reader interested in composition schemes can consult [108].

### 4.5 Related Efforts

This literature review is not the first effort in trying to extract systematic knowledge from the research being conducted between the two disciplines of software engineering and control theory.

Most of the survey work on the subfield of *adaptive software* focuses on architecture-based adaptation [83, 150], where MAPE loops are usually considered the main technique to design an adaptation strategy and can be coupled with additional knowledge to reason about the software and the environment.

Motivated by the need for formal guarantees in the design of self-adaptive systems, researchers started to explore the application of principles from control theory to adapt computing systems, introducing the notion of “dynamic feedback” [53]. Seminal research in this direction is documented in the book by Hellerstein et al. [79], which highlights the potential of control theory for the adaptation of computing systems. As a result, a number of authors have further investigated the interplay between control theory and software engineering.

A pioneering article that elaborates on the application of control theory to software servers to provide guarantees for adaptation is [4]. Based on that and on subsequent works, control theory was considered as an approach that can be used in software engineering for the design of software that modifies its behavior at
runtime providing formal guarantees about the mechanism used for the adaptation and about the goal satisfaction, whenever possible [24, 48].

While these studies can be useful in understanding the relationship between software engineering and control theory, they do not provide a comprehensive in-depth overview of the state of the art and they focus on adaptation at all the possible levels – as highlighted by the examples in Figure 4.1.

There have been a number of surveys in particular computing domains, for example [78] on mechanisms for performance management of Internet applications and [145] on quality-driven software adaptation using system properties derived from control theory to evaluate the usefulness of the adaptation. These surveys only investigated resource allocation and admission control, without delving into adaptation of the software behavior. A recent review of cloud service selection approaches did not identify any application of control theory for the adaptation of higher system layers in the cloud [140].

The work that is closely related to this survey is the systematic literature review on control-based adaptation of computing systems realized by Patikirikoral et al. [126]. The main result of that effort is a taxonomy that captures the characteristics of target and control systems, together with the types of validation performed to verify the effectiveness of the control mechanism. However, [126] does not distinguish between control-based adaptation at different layers of computing systems and treats low-level adaptation mechanism similarly to software adaptation. Low- and high-level adaptation are different in many aspects, the most important one being probably the availability of adequate physical models to guide the control design [98]. The analysis of low and high-level adaptation strategy lead to cluttered results that mix models, controllers, and guarantees of software adaptation with resource allocation, admission control, and hardware adaptation. From the results of this study it is therefore impossible to grasp the basic underlying principles that can be used for high-level software adaptation. Another problem of this survey [126] is the absence of data about a number of key characteristics and properties that are inherent to control theory. For example, the authors only collected data to classify system models based on their type – black box, first principle, queuing system –, while other essential model properties like linearity or non-linearity and discreteness versus continuity were not examined. The same limitation applies to actuators and controller purposes – regulatory action, optimal control, disturbance rejection. The classification of controllers provided by the authors mixes control-theoretic concepts. For example, PID, LQR and MPC, which are the controller types, are mixed with cascaded, decentralized and hierarchical control, which are approaches to compose multiple controllers of one of the mentioned types. Finally, the authors of [126] do not discuss the guarantees provided by the control-theoretical approaches, while formality is one of the main reasons to apply control theory [122, 14, 79, 24, 48].

In contrast to existing work, we perform a systematic literature review investigating control-theoretical software adaptation at the level of application software and supporting middleware services. We focus on adaptation based on classical or advanced control theory. This scope allows us to gain general insight and explore
the use of control theory as a foundation for the design, analysis and verification of adaptive software.

4.6 Research Method

To conduct our systematic literature review, we followed the guidelines described in [96]. In a first stage, the team defined a protocol to be used for the review. The protocol includes (a) research questions, (b) a search string to find relevant sources, (c) inclusion and exclusion criteria to determine if a document that was found with the given string is relevant or not, and (d) relevant venues to be used as data sources. In the remainder of this section, we discuss these key elements.

Given these elements, we performed two independent searches in the documents retrieved with the search string applied to the relevant venues and compared the results and resolved the ambiguities and discrepancies arisen.

Research Questions

We first formulated the overall goal of our literature review using the Goal-Question-Metric approach [16]:

Purpose: Understand and characterize

Issue: the use of control theory

Object: to adapt application software and supporting middleware services

Viewpoint: from the standpoint of a researcher.

As control-theoretical software adaptation only recently emerged as a research field and there is currently no good overview of the field, the primary aim of this review is to create such an overview. This overview will enable researchers to better compare and position specific contributions in the future.

We distilled the overall goal of the literature study in the following four research questions:

**RQ1:** What is the current state of research on control-theoretical adaptation of software at the application and middleware level?

**RQ2:** What are the model paradigms used for control-theoretical adaptation of software?

**RQ3:** What are the control strategies used for control-theoretical adaptation of software?

**RQ4:** What type of goals are achieved with control-theoretical adaptation of software and what kind of guarantees are provided?

RQ1 aims to provide a general overview of the state of the art in control-theoretical software adaptation. In particular, with RQ1 we can get insight in the trends of research on adapting software using principles from control theory. We
plan to provide a deep understanding of the motivations for the use of control theory, of the viewpoint taken in its use, and of the approaches used to assess the effectiveness of the control approach.

The other questions are aligned with the “three broad areas of challenges in applying control theory to computing systems” mentioned by Hellerstein et al. [79, p.24]. These areas are: constructing models of the target system and controller, designing the feedback controllers, and defining evaluation criteria to assess the results obtained.

Concretely, we formulated RQ2 to identify the models used for controlling software and their characteristics. RQ3 is related to the types of controllers and their different use, to the sensors and actuators applied, and to the methods used for building controllers. Finally, RQ4 helps us identifying the methods and metrics used to assess the effectiveness of the control solution and the guarantees provided.

### Document Sources

To select the sources used for our systematic literature survey, we followed the same procedure used for other systematic studies, such as [150, 70]. The procedure starts with identifying the document sources that are used in related surveys [150, 126, 48]. The sources are then refined by consulting with researchers from both the field of control theory and software engineering.

After following the mentioned procedure, we identified the main venues for publishing research in control theory, software engineering and adaptive systems. To ensure high quality and obtain solid data to answer the research questions, we excluded a number of venues based on two parameters: the Australian Research Council (ARC) ranking\(^1\) and the H-index\(^2\). Most of the included venues have high ARC rating (A*/A) and an H-index higher than 10. However, ranking alone is usually not conclusive. Therefore, we included a number of conferences and journals independent of their ratings because they are considered important in the respective communities.

In total, we included 41 venues: 15 journals and 26 conferences. For the journals we included 11 from control theory (CT), 3 from software/systems engineering (SSE), and 1 from adaptive systems (AS), see Table 4.3 for more detailed information. For the conferences, we included 4 from control theory, 17 from software/systems engineering, and 5 from adaptive systems, see Table 4.1 and Table 4.2.

### Search Strategy

Our search strategy is composed by six different steps.

---

1. ARC for journals: http://research.unsw.edu.au/excellence-research-australia-era-outlet-ranking
2. H-index for journals: http://www.scimagojr.com

For conferences: http://academic.research.microsoft.com/
Control venues: Google Scholar cat. Automation & Control Theory
<table>
<thead>
<tr>
<th>ID</th>
<th>Group</th>
<th>Venue</th>
<th>ARC</th>
<th>H-index</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICSE</td>
<td>SSE</td>
<td>International Conference on Software Engineering</td>
<td>A*</td>
<td>118</td>
</tr>
<tr>
<td>ICAC</td>
<td>SSE</td>
<td>International Conference on Autonomic Computing</td>
<td>B</td>
<td>32</td>
</tr>
<tr>
<td>DAC</td>
<td>SSE</td>
<td>Design Automation Conference</td>
<td>C</td>
<td>73</td>
</tr>
<tr>
<td>ICSM</td>
<td>SSE</td>
<td>International Conference on Software Maintenance and Evolution</td>
<td>A</td>
<td>56</td>
</tr>
<tr>
<td>ASE</td>
<td>SSE</td>
<td>Automated Software Engineering Conference</td>
<td>A</td>
<td>44</td>
</tr>
<tr>
<td>ESEC</td>
<td>SSE</td>
<td>European Software Engineering Conference</td>
<td>B</td>
<td>44</td>
</tr>
<tr>
<td>WADS</td>
<td>SSE</td>
<td>Workshop on Architecting Dependable Systems</td>
<td>n/a</td>
<td>30</td>
</tr>
<tr>
<td>VMCAI</td>
<td>SSE</td>
<td>Verification, Model Checking and Abstract Interpretation</td>
<td>B</td>
<td>30</td>
</tr>
<tr>
<td>WICSA</td>
<td>SSE</td>
<td>Working Conference on Software Architecture</td>
<td>A</td>
<td>25</td>
</tr>
<tr>
<td>CBSE</td>
<td>SSE</td>
<td>Symposium Component-Based Software Engineering</td>
<td>A</td>
<td>21</td>
</tr>
<tr>
<td>HASE</td>
<td>SSE</td>
<td>Symposium on High Assurance Systems Engineering</td>
<td>B</td>
<td>19</td>
</tr>
<tr>
<td>SEFM</td>
<td>SSE</td>
<td>Conference on Software Engineering and Formal Methods</td>
<td>B</td>
<td>18</td>
</tr>
<tr>
<td>ATVA</td>
<td>SSE</td>
<td>Symposium on Automated Technology for Verification and Analysis</td>
<td>A</td>
<td>14</td>
</tr>
<tr>
<td>QoSA</td>
<td>SSE</td>
<td>Conference on the Quality of Software Architectures</td>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>ECSA</td>
<td>SSE</td>
<td>European Conference on Software Architecture</td>
<td>n/a</td>
<td>8</td>
</tr>
<tr>
<td>FSE</td>
<td>SSE</td>
<td>International Symposium on the Foundations of Software Engineering</td>
<td>A</td>
<td>8</td>
</tr>
<tr>
<td>ESEM</td>
<td>SSE</td>
<td>Symposium on Empirical Software Engineering</td>
<td>A</td>
<td>n/a</td>
</tr>
</tbody>
</table>
The first step is the definition and validation of the search string to be used for automated search. This process started with pilot searches on IEEE Explore and the ACM Digital Library. We combined different keywords from software engineering and control theory that are relevant for our research questions. Based on the pilot searches, we defined the following search string, that was then applied to title and abstract.

\[(control \ OR \ controller \ OR \ controlling) \ AND \ (adaptive \ OR \ self-adaptive \ OR \ adaptation \ OR \ self- \ OR \ autonomic \ OR \ autonomous) \ [AND \ (software)]\]  

To validate the search string, we used a “quasi-gold standard” [163]. In particular, we manually searched through the proceedings of three known venues (TAAS, ICAC, and ICSE) during the past three years and found five studies that matched the selection criteria (discussed below). Then, we performed the automatic search in the proceedings of the same venues, using the search engines of the IEEE and ACM libraries. We refined the search string until the five studies were in the search results and the remaining number of the studies was minimal.

In the second step, we applied an automatic search using the previously defined search string. We use IEEE Explore, the ACM Digital Library and Google

---

3The additional keyword “software” is used only for control theory venues to improve the search results.
Table 4.3: Journals included in the search.

<table>
<thead>
<tr>
<th>ID</th>
<th>Group</th>
<th>Journal</th>
<th>ARC</th>
<th>H-index</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSE</td>
<td>SSE</td>
<td>Transactions on Software Engineering</td>
<td>A*</td>
<td>128</td>
</tr>
<tr>
<td>JSS</td>
<td>SSE</td>
<td>Journal of Systems and Software</td>
<td>A</td>
<td>72</td>
</tr>
<tr>
<td>TOSEM</td>
<td>SSE</td>
<td>Transactions on Software Engineering and Methodology</td>
<td>A*</td>
<td>59</td>
</tr>
<tr>
<td>Automatica</td>
<td>CT</td>
<td>Automatica</td>
<td>A*</td>
<td>85</td>
</tr>
<tr>
<td>TAC</td>
<td>CT</td>
<td>Transactions on Automatic Control</td>
<td>A*</td>
<td>82</td>
</tr>
<tr>
<td>TCST</td>
<td>CT</td>
<td>Transactions on Control Systems Technology</td>
<td>A</td>
<td>54</td>
</tr>
<tr>
<td>IJRCN</td>
<td>CT</td>
<td>International Journal of Robust and Nonlinear Control</td>
<td>A</td>
<td>41</td>
</tr>
<tr>
<td>CEP</td>
<td>CT</td>
<td>Control Engineering Practice</td>
<td>B</td>
<td>38</td>
</tr>
<tr>
<td>SICON</td>
<td>CT</td>
<td>SIAM Journal on Control and Optimization</td>
<td>A*</td>
<td>36</td>
</tr>
<tr>
<td>IJC</td>
<td>CT</td>
<td>International Journal of Control</td>
<td>A</td>
<td>33</td>
</tr>
<tr>
<td>CS</td>
<td>CT</td>
<td>IEEE Control Systems</td>
<td>B</td>
<td>24</td>
</tr>
<tr>
<td>SCL</td>
<td>CT</td>
<td>Systems &amp; Control Letters</td>
<td>n/a</td>
<td>41</td>
</tr>
<tr>
<td>ARC</td>
<td>CT</td>
<td>Annual reviews in control</td>
<td>n/a</td>
<td>27</td>
</tr>
<tr>
<td>CTA</td>
<td>CT</td>
<td>IET Control Theory &amp; Applications</td>
<td>B</td>
<td>39</td>
</tr>
<tr>
<td>TAAS</td>
<td>AS</td>
<td>Transactions on Autonomous and Adaptive Systems</td>
<td>B</td>
<td>26</td>
</tr>
</tbody>
</table>

Scholar\(^4\). The search is performed on the venues described in Section 4.6. For venues not included in the digital libraries, we manually downloaded and searched the proceedings. After the automatic search, we collected a total of 1512 papers.

In the third step, two researchers independently read the abstracts of all studies selected in the previous step and used the inclusion and exclusion criteria described in Section 4.6 to filter out irrelevant papers. Of the 1512 papers selected with the automatic string match, only 161 papers were advanced to the next stage.

In step four, we read the complete papers to make a final decision on their inclusion in the review. Conflicts were resolved during extensive discussion. We

\(^4\)The reference search string was adjusted to match the search features provided by different electronic sources (e.g., different field codes, case sensitivity, syntax of search strings). The search string for Google Scholar was adjusted to *controller adaptive software* as the engine only allows searching on title or full text of papers.
excluded a various number of papers because they were not relevant, and had 40 studies to analyze at the end of this step. As a fifth step, we applied snowballing. We checked the references cited by the selected papers and included them when appropriate. We increased the number of studies to analyze to 61 papers.

Finally, in the sixth step, we identified and removed similar versions of the remaining papers. For example, when we found a conference and a journal version of the same paper, we kept only the journal version, as it is considered more complete and accurate. The final list of primary studies for our literature review consists of the following 42 references: [98, 116, 55, 99, 50, 64, 63, 10, 65, 81, 66, 136, 127, 128, 125, 3, 90, 20, 5, 105, 104, 100, 58, 61, 59, 36, 118, 28, 27, 89, 88, 37, 162, 129, 144, 117, 38, 8, 137, 9, 92, 91].

Inclusion and Exclusion Criteria

We determined that a paper is approved for further analysis only when it satisfies all inclusion criteria and does not satisfy any of the exclusion criteria. We include studies that:

- Were published from January 2000 to June 2016. We used 2000 as starting date as adaptive systems have become subject of active research around that time [53, 4].

- Discussed the engineering of the adaptation strategy. The design or the implementation of the adaptation strategy or its parts must be included in the study.

- Matched the focus of the study, including adaptation of application software or high-level reusable middleware services, as shown in Figure 4.1.

- Applied classic or advanced control theory to design feedback loops, as shown by the grey area in Figure 4.2.

We excluded:

- Papers written in languages other than English.

- Tutorials, short papers, editorials because they do not contain sufficient data for our study.

Assessment of the Presentation Quality

Assessing the quality of the presentation – not necessarily related to the quality of the research – of the studies is important for the interpretation of the results. To assess the presentation quality, we collected six quality items for each study. The quality items are listed in Table 4.4. These items are based on the quality assessment method for research studies initially described in [56] and adjusted in [154]. For each quality item we assign a value of 2 if the authors provide an explicit description, 1 if there is a general description, and 0 if there is no description at all. The paper quality assessment score (max 12 points) is calculated by summing up the scores for every quality item.
**Table 4.4:** Quality items to assess the presentation quality of the studies.

<table>
<thead>
<tr>
<th>Q1: Problem definition of the study</th>
<th>2</th>
<th>Explicit problem description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>General problem description</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>No problem description</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q2: Problem context of the study</th>
<th>2</th>
<th>Explicit problem context supported by references</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>General problem context supported by references</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>No description of the context</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q3: Research design of the study</th>
<th>2</th>
<th>Explicit description of how the research was organized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>General words about the way the research was organized</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>No description of how the research was organized</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q4: Contributions/results of the study</th>
<th>2</th>
<th>Explicit list of the study contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>General words about the study contributions</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>No description of the study contributions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q5: Insights derived from the study</th>
<th>2</th>
<th>Explicit list of insights/lessons learned from the study</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>General words about the insights</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>No description of the insights derived from the study</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q6: Limitations of the study</th>
<th>2</th>
<th>Explicit list of the study limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>General words about the study limitations</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>No description of the study limitations</td>
</tr>
</tbody>
</table>
Table 4.5: Collected Data items.

<table>
<thead>
<tr>
<th>Item ID</th>
<th>Field</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Author(s)</td>
<td>Documentation</td>
</tr>
<tr>
<td>F2</td>
<td>Year</td>
<td>Documentation</td>
</tr>
<tr>
<td>F3</td>
<td>Title</td>
<td>Documentation</td>
</tr>
<tr>
<td>F4</td>
<td>Venue</td>
<td>Documentation</td>
</tr>
<tr>
<td>F5</td>
<td>Citations per year</td>
<td>Documentation</td>
</tr>
<tr>
<td>F6</td>
<td>Quality score</td>
<td>Documentation</td>
</tr>
<tr>
<td>F7</td>
<td>Engineering perspective</td>
<td>RQ1</td>
</tr>
<tr>
<td>F8</td>
<td>Motivation for CT</td>
<td>RQ1</td>
</tr>
<tr>
<td>F9</td>
<td>Validation</td>
<td>RQ1</td>
</tr>
<tr>
<td>F10</td>
<td>Assessment</td>
<td>RQ1</td>
</tr>
<tr>
<td>F11</td>
<td>Application domain</td>
<td>RQ1</td>
</tr>
<tr>
<td>F12</td>
<td>Claimed Generality</td>
<td>RQ1</td>
</tr>
<tr>
<td>F13</td>
<td>System model</td>
<td>RQ2</td>
</tr>
<tr>
<td>F14</td>
<td>Sensors and actuators</td>
<td>RQ3</td>
</tr>
<tr>
<td>F15</td>
<td>Triggers for adaptation</td>
<td>RQ3</td>
</tr>
<tr>
<td>F16</td>
<td>Controller type</td>
<td>RQ3</td>
</tr>
<tr>
<td>F17</td>
<td>Controller purpose</td>
<td>RQ3, RQ4</td>
</tr>
<tr>
<td>F18</td>
<td>Guarantees</td>
<td>RQ4</td>
</tr>
<tr>
<td>F19</td>
<td>Software qualities</td>
<td>RQ4</td>
</tr>
<tr>
<td>F20</td>
<td>Tradeoffs</td>
<td>RQ4</td>
</tr>
</tbody>
</table>

Extracted Data Items

Table 4.5 shows the data items that are extracted to answer the identified research questions. We here briefly explain the different items.

- **F1-F5**: These data items are used for documentation. For item F4 we additionally group venues into SSE (Software/Systems Engineering), CT (Control Theory) and AS (Adaptive Systems), as shown in Table 4.1. This data item is referred as F4.1.

- **F6**: Presentation quality score (on a total of 12), obtained as described in Section 4.6.

- **F7**: The engineering perspective taken by the authors of the study, which can be one of the following options: (a) *SE perspective*: The focus of these studies is on applying adaption to realize some quality requirements. The application
of control-theoretical principles to adapt software is not well elaborated. For example, controller guarantees are not analyzed or the software mathematical model is not explicitly presented. (b) **CT perspective:** The focus of these studies is control theoretical aspects; software is basically used as an application domain. There is less focus on typical software engineering aspects. (c) **Integrated perspective:** These studies employ principles from control theory to solve a software adaptation problem and exploit its mathematical foundation to analyze the system behavior and provide guarantees for quality goals.

- **F8:** Motivation for using control theory in a software system. The initial options are: formal guarantees, systematic approach, inefficiency of existing approaches. Additional options are derived during the review.

- **F9:** Validation setting is one of the following: academic effort, academic/industry collaboration, industrial effort, none.

- **F10:** The assessment approach used in the study. The initial options are: example application, simulation and discussion. In addition, we collect data about formal assessment (F10.1) which is one of the following: formal modeling, formal analysis, or none. By formal modeling we mean having a formal description of system model/controller, while formal analysis includes analysis of guarantees.

- **F11:** Applications domain for which adaptation is used or evaluated in the study. For example, e-commerce, tourism, video processing. The concrete application domains are derived during the review.

- **F12:** A boolean indicating whether the authors state a general applicability of the proposed approach.

- **F13:** The system model. Extracted data are divided into four sub-properties: (F13.1) model type, (F13.2) model linearity, (F13.3) time framework, (F13.4) model time dependency. For F13.1, a system can be denoted as (a) analytical, (b) grey box, or (c) black box. In an analytical model, the system is described by laws governing the behavior of that system (e.g., a Markov Chain). All model elements are known at design time (but parameters may change at runtime). With a grey box model, the system is not entirely known, a certain model based on both insight in the system and experimental data can be constructed. However, the model has a number of unknown free parameters that are estimated using system identification. In the black-box case, the system is considered unknown but can receive input and produces some output, that in principle comes from unknown functions. For F13.2, a model can be either (a) linear or (b) non-linear. In the linear case, the output is directly proportional to the input. In the non-linear case, this direct proportionality is not true. F13.3 can be either (a) discrete- or (b) continuous-time. In a discrete-time model, a system is modeled using difference equations, while continuous-time models rely on ordinary differential equations. As for F13.4, the model can be either (a) time-dependant, or (b) time-invariant. In the first case, the dependency on time is explicit. The output \( o \) is computed using a function \( f \) that depends on the input \( i \), on the state \( x \), and on time \( t \),
4.6 Research Method

\[ o(t) = f(i, x, t) \]. In the second case, the model describes the output at some time advancement, but the relationship does not contain time \( o(t) = f(i, x) \) and depends only on the state and the input.

- **F14**: Sensors and actuators. We separate collected data into: (F14.1) sensors: what is being measured during adaptation, (F14.2) actuators: the mechanism affecting software behavior to achieve the adaptation goals, (F14.3) triggers for adaptation: can be one of the following: stimulations from the environment, changes in requirements/goals, changes in the software itself.

- **F16**: The controller type used in the feedback mechanism. Options include PID, MPC, optimal, and others. In addition, we collect the data about: (F16.1) adaptivity of controller: adaptive or non-adaptive, (F16.2) composition scheme of multiple controllers, if applicable. Options include cascaded, hierarchical, and others.

- **F17**: The controller purpose with options: optimization, regulatory functions (setpoint tracking), disturbance rejection, or a combinations of these purposes [79].

- **F18**: Formal guarantees provided by the use of control theory and described in the study. According to [79], control theory can guarantee four main system qualities: stability, steady-state error, settling time, and maximum overshooting. Stability refers to the ability of the system to converge to a fixed point (as opposed to diverging – for example, accumulating requests in a buffer). Steady-state error refers to the difference between the fixed point to which the system converged to and the desired goal, given to the controller. The settling time of a controller is a measure of how quickly the controller is able to reach the fixed point, when it exists. Finally, the maximum overshoot determines how much the maximum difference between the measured value and the objective will be, during the transient phase. A graphical summary of these properties can be seen in Figure 4.6. Additionally to the properties mentioned in [79], a number of studies discuss the guarantees of systems with respect to robustness. Robustness is the ability of the system to return to the steady-state in case of model inaccuracies or perturbations and disturbances. We also collect data about experimentally verified guarantees (F18.1). The difference with the data extracted in F18 (formal guarantees) is that the evidence is based on data that is collected from experiments.

- **F19**: Software qualities that are affected by adaptation and described in the study. We use the specification of qualities described in the ISO/IEC 9126-1 standard\(^5\). According to [150], the software engineering approaches mostly concentrate on: (a) performance, the ability of the software to achieve a desired value for qualities like throughput and response time; (b) efficiency, the extent to which the software uses the appropriate resources under stated conditions and in a specific context of use; (c) reliability: the capability of software to maintain its level of performance under stated conditions for a

period of time; (d) other: other software qualities such as scalability, flexibility, usability, security, and portability.

- F20: Concerns that can be degraded as a consequence of improving other concerns. This can be one or several guarantees listed in F18 and/or qualities listed in F19.

### 4.7 Result Analysis

This section summarizes the data we collected from the identified 42 studies and presents an analysis of the results. We use descriptive statistics and plots for visual presentation of the results. We first present demographics information and presentation quality assessment. Then, we answer the research questions stated in Section 4.6 based on the collected data.

#### Demographics

Figure 4.7 shows the frequency of primary studies per year (item F2).

Although we looked at papers from the past 15.5 years, we observed that 72% of primary studies were written in the last 5.5 years. This indicates that there is a growing interest in research on control-theoretical design of software. Several authors have argued that one important factor for this growing interest is the mathematical foundation of control theory that provides a solid basis for guaranteeing the adaptation goals under uncertainty [127, 10, 65, 137].

Our review revealed that the publication of primary studies is scattered over different venues: 25 studies were published at software/systems engineering venues, 7 at control theory related venues, 6 at adaptive systems venues, and 4 at venues
with other subjects. The only venue that published more than three of the studies was the Journal of Systems and Software with 6 studies. Having the majority of studies published at software/systems engineering venues, we can conclude that there is more interest in the software/systems engineering community in exploring the application of principles from control theory to realize adaptation of software as from the control theory community in applying novel research results to software applications.

Table 4.6: Studies with minimum 10 citations per year.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Reference</th>
<th>Cit./year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Control of Web Server</td>
<td>[5]</td>
<td>46.2</td>
</tr>
<tr>
<td>Brownout Paradigm</td>
<td>[98]</td>
<td>12.5</td>
</tr>
<tr>
<td>Push Button Methodology</td>
<td>[65]</td>
<td>11.5</td>
</tr>
<tr>
<td>DYNAMICO Reference Model</td>
<td>[144]</td>
<td>10.7</td>
</tr>
</tbody>
</table>

We also sorted studies according to the number of citations per year (item F5). Table 4.6 shows the primary studies with minimum 10 citations per year.

Key insights from demographic:

- The interest in the research on control-theoretical adaptation raised significantly in the last 5.5 years.
- The publication of the primary studies is scattered over different venues.

Presentation Quality

The results of presentation quality assessment of the primary studies (item F6, Figure 4.8) show that the majority of the studies provide an in-depth description of the problem and the problem context, and most studies give a sufficiently clear description of contributions and insights. However, many studies do not describe the research design (methods, different steps, etc.) and lack a discussion of limitations.
of the proposed approach. This seems to be a general trend as similar results have been reported in other secondary studies and other domains, see e.g., [150, 70]. Nevertheless, the overall average score of 7.3 out of 12 points indicates a good quality of reporting in the studies, supporting the validity of the extracted data and the conclusions derived from them.

The particular limitations reported in the primary studies are summarized in Table 4.7. Notably, most of the limitations concern the applicability of the proposed adaptation mechanism (pre-conditions, redundancy, complexity). Only a few of the primary studies explicitly report threats to validity of the conducted study, such as internal validity, construct validity, and external validity.

Key insights from presentation quality:

• Most of the primary studies provide a comprehensive description of problem and context, but lack a discussion of research design and limitations.

RQ1: Control-Theoretical Software Adaptation

To answer the first research question (what is the current state of research on control-theoretical adaptation of software at the application and middleware level?), we used data items F7-F12. Figure 4.9 provides an overview of the results. The engineering perspective taken in the studies varied (item F7, Figure 4.9a). 11 studies took a software engineering perspective. In these studies, particular attention was given to typical software engineering aspects, such as software qualities, design, testing, and similar concerns. The application of control theory to realize adaptation of software was not well elaborated. For example, guarantees provided by control theory were not analyzed and the software model and controller structure was not well defined. 10 studies took a control theoretic perspective. The focus of these studies contrasts to the software engineering perspective: attention was given to the formal part of the adaptation, the studies included an in-depth mathematical analysis of the model/con-
Table 4.7: Limitations reported in the primary studies

<table>
<thead>
<tr>
<th>Limitations</th>
<th>Primary Studies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires specific conditions to function</td>
<td>[128, 125, 104, 89, 129, 66, 5, 9]</td>
</tr>
<tr>
<td>(pre-conditions)</td>
<td></td>
</tr>
<tr>
<td>Requires additional computation resources or tools</td>
<td>[128, 125, 5, 58]</td>
</tr>
<tr>
<td>(redundancy)</td>
<td></td>
</tr>
<tr>
<td>Not applicable in some cases/systems</td>
<td>[128, 104, 27, 89, 129, 10, 66, 5, 137]</td>
</tr>
<tr>
<td>External validity: generalising findings requires</td>
<td>[125, 27, 58, 36, 137]</td>
</tr>
<tr>
<td>extra effort</td>
<td></td>
</tr>
<tr>
<td>Complexity of the proposed approach</td>
<td>[125]</td>
</tr>
<tr>
<td>Not able to handle new requirements</td>
<td>[8]</td>
</tr>
</tbody>
</table>

troller. Software, in this case, was used as an application domain, typical software engineering aspects were not well elaborated. The remaining 21 studies [98, 99, 117, 38, 36, 5, 127, 28, 63, 61, 10, 81, 66, 65, 136, 8, 89, 88, 137, 9, 92] took an integrated perspective. These studies employed both software engineering and control theoretic aspects to realize adaptation of software.

The motivations to apply principles from control theory for adapting software varied (item F8, Figure 4.9c). The main motivations documented in the primary studies were formal guarantees, maturity (“systematic approach” and “solid foundation”), and effectiveness of control theory. These results support the rationale (discussed in Section 4.7) that software engineers are exploring new well-grounded approaches for engineering self-adaptive software driven by the need for guarantees. Note that 27% of the studies did not provide any motivation for applying control theory to software adaptation. We could not derive any conclusive data why the authors of these primary studies have not provided a motivation. Furthermore, there is no dominating trend in the motivations that are reported in the other studies (see Figure 4.9c). The motivations for applying control-theoretical adaptation of software may be an interesting topic for further investigation.

Validation of the research (item F9), except two industrial studies [92, 91], was based on academic efforts. As research of control-theoretical software adaptation is still in its early stages, most of the results have not yet found their way to practice. The most used assessment methods (item F10, Figure 4.9b) were example application, followed by simulation. These results are in line with the results presented in [126]. In 38 out of 42 primary studies formal modeling or analysis is conducted (item F10.1, Figure 4.9b). This is not surprising and confirms the appreciation of the formal underpinning of control theory to realize control-theoretical software adaptation. The concrete types of guarantees that are analyzed in the primary studies are discussed in Section 4.7.

The most popular application domains in the primary studies (Figure 4.9d) were web applications (E-commerce) and video/image processing software. The most
(a) F7: Engineering perspective – F12: Claimed generality.

(b) F10 Assessment and F10.1: Formal Assessment.

(c) F8: Motivation for using control theory.

(d) F11: Application domain.

**Figure 4.9:** Results for data items required to answer RQ1.
used E-commerce applications were a flight reservation system described in [127] and the RUBiS benchmark\(^1\). Three studies used general web applications that show static content to user [5, 105, 104], while one study used recommender systems [162]. The applications in the video/image processing domain can be divided in different groups: object recognition [100, 58], video streaming [117, 20], video encoding [38, 65, 81], image and signal processing [64, 3].

Two abstract design/technology paradigms (service-based system and search engine) were included as six studies used these paradigms without describing a concrete application domain, e.g. [61, 59, 27] (Figure 4.9d between the dotted and full horizontal lines).

Finally, six studies applied principles from control theory not directly to adapt a running software application, but to support software development (Figure 4.9d). In particular, these studies applied control theory to calculate the human resources required for testing a software product [36, 118], to determine the quality of tests [28, 27], to select the appropriate types and number of test cases in order to minimize the number of software defects, to optimally distribute the development effort between construction and debugging [89], and to analyze the system lifetime based on the amount of development effort [88].

We observed that 18 primary studies stated general applicability of the proposed approach (item F12). It is notable that 7 out of 12 studies with software engineering focus (Figure 4.9a) proposed a generally applicable framework or methodology. On a contrary, only one study with a control theory perspective claimed the general applicability of the proposed approach [3]. These results support the tendency of research in the control engineering community to develop specific solutions for concrete problems, while in the software engineering community it is more common to aim for generally applicable solutions [71, 102, 156]. One of the main reasons to build controllers for specific problems in control theory is that generality comes with a tradeoff: generality of a controller typically implies some decrease in performance or robustness objectives [74]. Nevertheless, control theory offers a number of generic control structures (or patterns of controllers) and engineering techniques that enable these control structures to automatically adjust to specific scenarios [12].

As a side note, it is important to mention that the evidence for the general applicability of the proposed approaches in most of the primary studies is limited to the evaluation of a few examples or provided in form of discussion.

\(^1\) Rice University Bidding System: http://rubis.ow2.org
RQ1: Control-Theoretical Software Adaptation

- The main motivations to use control theory in software adaptation are the maturity of the field and its formal foundation as a basis to provide guarantees.
- The most used application domains for control-theoretical software adaptation are E-commerce and video/image processing.
- Assessment of research contributions is based on (simple) example applications and simulations. There is a need for involving industry partners to evaluate control-theoretical solutions in practical settings.
- The studies with a software engineering focus typically propose a generally applicable methodology/framework, while studies focusing on control theory solve specific problems.

RQ2: Software Models

To answer the second research question (what are the model paradigms used for control-theoretical adaptation of software?), we used data item F13, see Figure 4.10.

We observed that different types of system models are used, but the dominating type is a linear, time-invariant, discrete grey-box model that is built using system identification techniques. The studies apply linear grey-box models for three reasons. First, these models can be easily designed, see for example [65, 63]. Black or grey-box models are preferred as it is often difficult to create a detailed analytical model of software since it is not governed by physical laws. And even if such model can be created, it may become inaccurate after the first software update. Moreover, the parameters of an analytical model must be updated at runtime to deal with changing operating conditions. Second, black or grey-box models offer a generic solution to system modeling. Whereas at the infrastructural layer CPU cores, memory and virtual machines can be easily abstracted for many systems types, at the software level it is problematic (or challenging) to find general el-
Table 4.8: F13: System model.

<table>
<thead>
<tr>
<th>Behav. Model</th>
<th>Specific Model</th>
<th>Primary Studies</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analytical</td>
<td>Markov model</td>
<td>[63, 64, 28, 27]</td>
<td>Model used for stochastic systems for which future states depend only on the current state.</td>
</tr>
<tr>
<td></td>
<td>Queuing network</td>
<td>[20, 10]</td>
<td>System is represented as a network of queues, which is evaluated analytically.</td>
</tr>
<tr>
<td></td>
<td>Custom</td>
<td>[59, 89, 88, 100, 58, 92, 91]</td>
<td>Custom analytical models used for object recognition [100, 58], software development process [89, 88], search algorithm [59], ads [92, 91].</td>
</tr>
<tr>
<td></td>
<td>Linear model</td>
<td>[98, 116, 55, 99, 50, 65, 136, 81, 66, 137]</td>
<td>Model of the form: (u(k+1) = \alpha \times \eta(k) + d(k)), where (u(k+1)) is the system output, (\eta(k)) is the actuation signal, (\alpha) is a coefficient, (d(k)) is a disturbance acting on the output. (u(k+1)) and (\eta(k)) are case-specific; coefficient (\alpha) is calculated during system identification based on a series of experiments [98, 65, 136] or using a controller [81, 66]. In some cases [65, 136] (d(k)) is removed from the model, while (\alpha) is updated during system operation to cope with system dynamics.</td>
</tr>
<tr>
<td>Grey box</td>
<td>Hammerstein-Wiener</td>
<td>[128]</td>
<td>Model that combines a non-linear block that captures the system non-linear behavior with a linear block responsible for all remaining system dynamics.</td>
</tr>
<tr>
<td></td>
<td>Multi-Model Switching</td>
<td>[125]</td>
<td>Models of different types that can interreplace each other during operation depending on the system goals.</td>
</tr>
<tr>
<td></td>
<td>Custom</td>
<td>[5, 104, 36, 118, 127, 38, 3, 90, 8, 162, 9]</td>
<td>Custom grey box models used for web server utilization [5, 104], software testing process [36, 118], resources allocation between software components [127, 38], recommender system [162], component interactions at the application layer [3, 90].</td>
</tr>
<tr>
<td>Black box</td>
<td>Custom</td>
<td>[105, 117]</td>
<td>Models used with the aim to achieve generality of the proposed approach.</td>
</tr>
<tr>
<td>N/A</td>
<td>Not specified</td>
<td>[61, 144, 129]</td>
<td>No specification of the concrete model being used.</td>
</tr>
</tbody>
</table>
ements that can be modeled. Each middleware software or each application has its own technology- and domain-specific software elements. Third, as stated in [65, 66], although linear grey-box models are not as accurate as complex non-linear models at design time, they are more effective at runtime due to a low level of complexity and a higher degree of guarantees that can be obtained using them. A common view on using a linear grey-box model is that as long as the model captures the general system dynamics, the inherent non-linearities of the system can be compensated by endowing the feedback controller with an adaptation or online model update mechanism [125, 105]. Table 4.8 provides an overview of models used in different studies.

Although most studies refer to the complexity of software systems and their non-linear behavior, there are only 11 studies that look at software as a non-linear system [27, 64, 28, 89, 88, 59, 128, 3, 90, 92, 91]. It is notable that most of the non-linear models are analytical (Figure 4.10a). An explanation for this is that the identification of non-linear models is extremely challenging in terms of engineering effort; and there are almost no tools available to support the identification of non-linear models [106, 69].

When software applications undergo sudden changes in their behavior at runtime (for example a component failure), we observed two types of reactions in the primary studies: (1) updating model parameters [58, 65, 66, 136, 36, 118, 27] or even switching the model [125], and (2) updating parameters of the control law, which means using adaptive or model predictive control (further discussed in the following Section). In some cases, an update of the model may be followed by an update of control law as well [65, 136, 27]. Other approaches use a separate linear corrector to compensate for model changes [118], allow human operators to make a decision [36], or completely change the control law [58, 125].

**RQ2: Software Models**

- Linear, time-invariant, discrete grey-box models are mostly used in control-theoretical software adaptation.
- Although most of the authors discuss complexity and non-linear behavior of software, only 11 out of 42 primary studies employed non-linear models, most of which are analytical.
- Eight primary studies deal with behavioral changes at runtime by updating model parameters.

**RQ3: Control Strategies**

To answer the third research question (what are the control strategies used for control-theoretical adaptation of software?), we look at: monitoring mechanisms (sensors), effecting mechanisms (actuators), triggers for adaptation and controller types.

**Sensors:** When extracting data about monitoring mechanisms (sensors) and effecting mechanisms (actuators), we observed that the actual implementation of sensors (for example how the values are technically measured) and actuators (for
Table 4.9: F14.1: Sensors.

<table>
<thead>
<tr>
<th>Sensor Type</th>
<th>Monitored Variables, the variables are representative examples</th>
<th>Primary Studies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software utility</td>
<td>Probability of correct object recognition</td>
<td>[100, 58]</td>
</tr>
<tr>
<td></td>
<td>Software response to requests</td>
<td>[50, 127, 128, 125, 136, 81, 66, 117]</td>
</tr>
<tr>
<td></td>
<td>User perceived latency of application</td>
<td>[98, 116, 55, 99]</td>
</tr>
<tr>
<td></td>
<td>Video quality and processing speed</td>
<td>[38, 65, 3, 90]</td>
</tr>
<tr>
<td></td>
<td>Profit gained from the application</td>
<td>[88, 129]</td>
</tr>
<tr>
<td>Software inefficiency</td>
<td>Percentage of software failures</td>
<td>[64, 63, 65, 66, 136]</td>
</tr>
<tr>
<td></td>
<td>Detection of software defect</td>
<td>[118, 28, 27]</td>
</tr>
<tr>
<td></td>
<td>Number of errors in software application</td>
<td>[36, 89, 129]</td>
</tr>
<tr>
<td>Resource utilization</td>
<td>Energy consumption</td>
<td>[81, 66]</td>
</tr>
<tr>
<td></td>
<td>Cost of using external services</td>
<td>[66]</td>
</tr>
<tr>
<td></td>
<td>Bandwidth</td>
<td>[5, 105, 104]</td>
</tr>
<tr>
<td></td>
<td>CPU usage</td>
<td>[37, 38]</td>
</tr>
<tr>
<td></td>
<td>Memory usage</td>
<td>[117, 37]</td>
</tr>
<tr>
<td>System utilization</td>
<td>Length of requests queue</td>
<td>[90, 20, 10, 99, 50]</td>
</tr>
<tr>
<td></td>
<td>Amount of new user registrations</td>
<td>[129]</td>
</tr>
<tr>
<td></td>
<td>Data to be processed by the system</td>
<td>[162]</td>
</tr>
<tr>
<td></td>
<td>Request arrival rate</td>
<td>[5, 105, 104, 3, 50]</td>
</tr>
</tbody>
</table>

Figure 4.11: F14.1: Sensors.

example how the actuation mechanisms implement changes of the application) are discussed in only 7 of the 42 primary studies [5, 104, 98, 66, 61, 144, 38]. [5, 104] describe how bandwidth and request rate of an Apache Web server are measured, [98] employs PHP scripts to control the amount of optional content served to users and calculate user perceived latency, and [61] compares the influence of actuator realizations on the output of the target software.

In the rest of the primary studies, the authors refer to sensors and actuators as the monitored variables and variables effecting the application respectively. Con-
sequently, we analyze only these variables in this literature review (and refer to
them as sensors and actuators), due to the lack of data concerning the actual im-
plementation of sensors/actuators in the primary studies.

We observed the use of various types of sensors in the primary studies (item
F14.1, Figure 4.11 and Table 5.1). The sensors can be classified in two main
classes: sensors that monitor the software that is subject of adaptation and sen-
sors that monitor elements that are external to the software application. We further
distinguish two types of sensors that monitor the software application: those that
monitor software utility and those that monitor software inefficiency. Sensors that
monitor software utility measure the usefulness of the software to achieve its goals,
such as the quality of video and the profit gained from the software application.
Sensors that monitor software inefficiency measure the lack of ability of the soft-
ware to achieve its goals, such as detection of software defects and errors in the
software application. We also distinguish two types of sensors that monitor ele-
ments external to the software application: those that monitor resource utilization
and those that monitor system utilization. The sensors that monitor resource uti-
lization measure the amount of resources consumed by the software application
to realize its goals, such as energy consumption and memory usage. Sensors that
monitor system utilization on the other hand measure the degree of load on the
application, for example as the length of request queues or the request arrival rate.
Table 5.1 lists other examples of the different types.

The first class of sensors – those that monitor software utility and software in-
efficiency – are specific to control-theoretical software adaptation. These sensors
have to be implemented by the software application or middleware services, for
example using supporting functionality (framework API, component model, pro-
gramming abstractions, and similar ones) or through a dedicated software inter-
face. The second class – sensors that monitor elements that are external to the
software application – are conventional types of sensors that are commonly used
in control-based adaptation of computing systems at lower levels of the technology
stack (low-level middleware and resources).

Control-theoretical software adaptation requires two types of sensors that re-
spectively monitor the software application and the execution environment. These
correspond to the types of sensors that are typically required for architecture-based
adaptation of application software. Two studies that elaborate on this are [144]
and [156].

Actuators: As for the actuators, we observed that the studies use a wide va-
riety of effecting mechanisms to realise adaptation (item F14.2, Figure 4.12 and
Table 4.10). We identified four main types of actuators that operate at different lev-
els of granularity: parametric, component, and mode adaptation, and architecture
reconfiguration.

Parametric adaptation refers to changing the values of variables of the applica-
tion software or supporting middleware services. These types of actuators are
typically domain-specific; examples are the degree of video compression and the
length of a queue with pending requests that need to be processed. Component
adaptation refers to changes at the level of software components, such as the load
of services and the degree of parallelism that components process requests. Mode
Table 4.10: F14.2 Actuators

<table>
<thead>
<tr>
<th>Actuator Type</th>
<th>Changed Variable, the variables are representative examples</th>
<th>Primary Studies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parametric adaptation</td>
<td>Length of a queue with pending requests [99]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Degree of video compression [65]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Quality parameters of a filter [81, 38]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Parameters to enhance the quality of testing [36, 118]</td>
<td></td>
</tr>
<tr>
<td>Component adaptation</td>
<td>Load of software services [64, 63, 65, 66, 136, 127, 128, 125]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Distribution of incoming requests [3, 90]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Test case of application [28, 27]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Degree of parallelism to process requests [117]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of service instances [20]</td>
<td></td>
</tr>
<tr>
<td>Mode change</td>
<td>Increment/decrement of content being served [98, 116, 55, 99, 50]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Change in search strategy [59]</td>
<td></td>
</tr>
<tr>
<td>Mode switch</td>
<td>Video buffering scheme [5, 105, 104]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Quality of content representation of website [65]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Preference given to each service level [10]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Operating mode of system [81, 38]</td>
<td></td>
</tr>
<tr>
<td>Architecture reconfiguration</td>
<td>Components change to handle variations in the task load [37]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Modules selected for execution to deal with changing goals [129]</td>
<td></td>
</tr>
</tbody>
</table>

adaptation refers to a variation in the mode of operation, which can be either mode change or mode switch. An example of a mode change is an increment in the quality of content that is being served by a video application; an example of mode switch is an alteration of the buffering schema of a video application. Finally, architecture reconfiguration refers to a runtime adaptation of the architectural structure or behavior of the application. We only observed two instances of this type of effecting mechanism: changing components to handle variations in the task load and selecting modules for execution to deal with changing goals.

As the actuators directly effect the application software and/or higher-level middleware services, they are all specific to control-theoretical software adaptation. Similar to the implementation of sensors, actuators can be implemented by supporting functionality (framework API, component model, and similar) or through a dedicated software interface. A particular aspect of effecting mechanisms is ensuring locality and consistency of the software adaptation. This means adapting the system properly without stopping or disturbing the operation of the parts of the system unaffected by the adaptation, which is more challenging for coarse-grained
types of adaptations, such as architecture reconfigurations. A typical approach to handle this is by adapting the system or parts of it in quiescent states [101]. We noticed that consistency of adaptation is to a large extent ignored in the primary studies of the survey. A related aspect of effecting mechanisms is that some adaptations may require more invasive changes, such as a partial or even complete reboot of the software system. Such kinds of adaptations are critical for controllers with a short adaptation period. A possible approach to address this aspect is suggested in one primary study that takes a software engineering perspective [104]. In this study, the authors encourage engineers to make sensors and actuators modifiable at runtime. Two other primary studies address this aspect by taking into account the controller overhead [38, 105]; a fourth study deals with it by minimizing the number of system reconfigurations [117].

We also checked whether there are any correlations between sensors/actuators and the system model (Figures 4.13 and 4.14). The analysis results give some indication that software utility sensors are the dominating type of sensors used, in particular for linear grey-box and time invariant models. Resource utilization is not used in analytical and non-linear models. Parametric adaptation and component adaptation are the dominating type of used actuators. Parametric adaptation is particularly preferred in analytical and continuous models; component adaptation is the preferred actuator for discrete, time invariant models. However, as the figures show, the data for both sensors and actuators is scattered over different model elements, so it is difficult to derive clear conclusions.

Finally, we gathered data about the triggers for adaptation (item F15). In 36 out of 42 primary studies adaptation is triggered by changes in the environment. In 29 primary studies adaptation is also triggered by changes in requirements. Only 11 studies present experiments with changing requirement at runtime [64, 63, 65, 81, 66, 136, 58, 59, 117, 37, 137], and only a single study [58] supports removing or adding new requirements on the fly. Finally, in 7 studies, adaptation is triggered by changes in the software itself. These studies are mainly related to software development and testing, where software is often the only source that provides feedback.

Controllers: The results for the data extracted for controller types (item F16, Table 4.11) shows that 5 types of controllers have been used for control-theoretical adaptation in the primary studies. The dominant type of controller is the PID controller (50% of the primary studies). While being the most applied type of con-
Table 4.11: F16 Controller type.

<table>
<thead>
<tr>
<th>Category</th>
<th>Specific Controller</th>
<th>Primary Studies</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID</td>
<td>Proportional-integral-derivative</td>
<td>[59, 129]</td>
<td>A classical controller that is easy to implement and tune. Consists of 3 components (P, I, D) responsible for different controller characteristics [14]. In 13 out of 18 studies, PID controllers are also adaptive as it helps to compensate for inaccuracy and errors in the system model [98, 63, 65]</td>
</tr>
<tr>
<td></td>
<td>Proportional-integral</td>
<td>[116, 98, 55, 99, 65, 136, 125]</td>
<td>Proportional [10, 36], Integral [61, 92]</td>
</tr>
<tr>
<td></td>
<td>Proportional Integral</td>
<td>[10, 36]</td>
<td></td>
</tr>
<tr>
<td>MPC</td>
<td>Model predictive</td>
<td>[127, 128, 118, 117, 38, 8, 9]</td>
<td>Controller that uses a system model to predict its future behavior and selects adaptation actions that minimize the cost for achieving this behavior.</td>
</tr>
<tr>
<td></td>
<td>Limited lookahead (LLC)</td>
<td>[90, 20, 3]</td>
<td>Controller that is conceptually similar to MPC: creates a set of future system states up to a certain horizon and selects a trajectory between these states such that its cost is minimal.</td>
</tr>
<tr>
<td>Feed-forward</td>
<td>Pure feed-forward</td>
<td>[162]</td>
<td>Controller that computes adaptation actions based on the system model; the system output is not taken into account, i.e., there is no feedback.</td>
</tr>
<tr>
<td></td>
<td>Feedback+ feedfoward</td>
<td>[116]</td>
<td>Applied in a single study, where feedforward and feedback controllers are paired and compared to other types of controllers.</td>
</tr>
<tr>
<td>Optimal</td>
<td>Custom optimal</td>
<td>[28, 27]</td>
<td>The goal of this controller is to minimize/maximize a cost function subject to certain constrains, e.g. maximize performance using a pool of limited resources.</td>
</tr>
<tr>
<td></td>
<td>Bang-bang</td>
<td>[89, 88]</td>
<td>This controller is also known as on-off controller because the control signal can take only two values, e.g., 0 or 1.</td>
</tr>
<tr>
<td>Deadbeat</td>
<td>Deadbeat</td>
<td>[81, 116]</td>
<td>Controller created using a pole placement technique [14, 79].</td>
</tr>
</tbody>
</table>
A Systematic Literature Review on Control-Based Software Adaptation

troller in the primary studies, it is less dominant as in industrial practice where PID controllers are used in around 90% of the control applications. MPC is used in 26% of the primary studies. MPC is the preferable choice for systems with multiple objectives. Other types of controllers used are Feedforward (5%), Optimal (10%) and Deadbeat (8%). Table 4.11 gives additional information about the controller types with examples how they are used in primary studies. It is notable that 4 primary studies do not specify a concrete controller, but instead refer to “any kind of feedback mechanism that makes a software fulfill its requirements” [64, 144, 100, 37]. We also underline a specific case of adaptation, where multiple feedback loops at different levels of computing systems interact with each other to address the adaptation goals. This case mostly occurred in primary studies that apply hierarchical control (e.g., [55, 99, 3, 90]), where a higher level controller of a software application provided goals for lower level controllers that manage resources such as CPU and memory.

Regarding the adaptivity of controllers (Figure 4.15a), in 13 out of 18 studies, PID controllers are also adaptive as it helps to compensate for errors that may result from the linearization in the modeling phase [98, 63, 65]. The other types of controllers used in the primary studies are mostly non-adaptive.

The data extracted for Controller purpose (item F17, Figure 4.15b) shows that PID is the preferred solution for regulatory control (setpoint tracking) and disturbance rejection. However, PID controllers do not scale easily, so their use is typically limited to single-input, single-output systems. The need to support adaptation for multiple objectives (for example: performance and failure rate), while functioning under constraints (like resource limitations) or requiring the system to optimize for some parameter (like minimizing the operational cost), led to the use of model predictive and optimal control [127, 38]. A well-known drawback of optimal controllers is that they are sensitive to modeling errors and runtime disturbances. This can be also observed in Figure 4.15b where optimal controllers are used solely for optimization purpose.

An interesting topic for analysis are possible correlations between controller types and system models. We observed the following tendencies (Figure 4.16):
4.7 Result Analysis

- 15 out of 17 primary studies use PID controllers with linear models. In addition to the complexity of building or identifying non-linear models, PID is not very effective in controlling processes that are non-linear and time-invariant [164, p.52]. A common practice from industrial control is combining a complex controller with a simple linear time-invariant model, and this approach seems to be adopted for software adaptation as well.

- All 10 studies with MPC controllers use discrete time-invariant models, 8 of which are grey-box models. The motivation for using discrete time-invariant models is similar to the use of linear models combined with PID control: it is a simple model to work with. Hence, it is preferred over complex non-linear or adaptive models. The motivation to combine grey-box models with MPC is based on the adaptation requirements of the software systems under study, which often have multiple inputs and outputs. As it is challenging to build a model of such systems without identification, a grey-box model is a preferred choice.

- The 4 primary studies focusing on optimal control used non-linear time-invariant analytical models. The motivations for using optimal control are similar to MPC, so it is not surprising to see a preference for time-invariant models. Nevertheless, the fact that all 4 studies use non-linear analytical models is surprising, and it worthwhile to see whether future studies will confirm this trend.

Finally, we looked at the composition of multiple controllers into a single feedback mechanism. The extracted data yields the following insights:

- Six of the primary studies apply hierarchical control. In 4 of these studies [55, 99, 3, 90], a high-level controller solves global software adaptation tasks and provides input for controllers at a second level that solve intermediate tasks and provide input for controllers of lower level that solve local adaptation tasks. A reversed two-level hierarchical control approach is studied in [136, 137], where multiple controllers at the top level provide inputs to a single controller at the bottom level.

- Two studies apply switching control [125, 58], where different control laws interchange with one another, depending on the actual software adaptation tasks.

![Figure 4.14: Relation between F14.2 Actuators and F13 System model.](image-url)
Two studies apply cascaded control [81, 66], where the output signal of a high level controller becomes an input for a lower level controller.

Finally, one study applies cooperative control [117], where multiple controllers work in parallel, contributing to achieve a global software adaptation task.

**RQ3: Control Strategies**

- Software adaptation requires specific sensors for measuring software utility and software inefficiency (along with conventional sensors to measure elements at lower levels of the technology stack and environment).

- The actuators directly effect the application software and/or higher-level middleware services, hence they that are all software-adaptation specific. Consistency of adaptation is largely ignored in the primary studies.

- PID and MPC are the dominating types of controllers used in software adaptation. The use of PID (50% of studies) is not as dominant as in current industrial practice.

- Studies using PID control, prefer to combine this with linear models, while studies that use MPC control prefer discrete time-invariant grey-box models.

- PID control is mostly used for regulatory functions and disturbance rejection in single-input, single-output systems. MPC and optimal control is mostly used to achieve optimality in systems with multiple goals.

**RQ4: Goals and Guarantees**

To answer research question four (what type of goals are achieved with control-theoretical adaptation of software and what kind of guarantees are provided?) we used data items F17-F20. The data extracted for software qualities (item F19,
4.7 Result Analysis

Figure 4.16: Relation between F16 Controller type and F13 System model.

Figure 4.17) shows that the primary focus is on performance, efficiency, reliability, and business value\(^2\) of the application.

The data extracted for guarantees (item F18) shows that 13 studies provide formal guarantees for required properties (item F18), while 13 primary studies provide empirical evidence for guarantees of required properties (item F18.1). Table 4.12 provides an overview of the different types of guarantees. Each type is illustrated with examples from studies that provide formal guarantees and studies that provide empirical evidence for guarantees.

The extracted data for quality tradeoffs (item F20) shows that most of the primary studies do not mention any tradeoffs. Only 3 primary studies consider tradeoffs between software qualities, namely, performance versus accuracy or reliability [3, 162, 89]. Seven studies discuss the tuning of a controller to trade different guarantees, typically robustness for settling time [98, 50, 65, 64, 10, 129, 137].

An interesting topic of analysis is the correlation between software qualities and achieved guarantees. Unfortunately, most studies do not provide a clear de-

\(^2\)Business value refers to the profit earned with the application.
Table 4.12: F18 Formal guarantees

<table>
<thead>
<tr>
<th>Guarantee</th>
<th>Formally Analyzed</th>
<th>Achieved by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Settling time</td>
<td>[98, 64, 10, 65, 66, 136, 137]</td>
<td>Analysing the pole of the controller.</td>
</tr>
<tr>
<td>Overshoot</td>
<td>[10, 66, 136, 137]</td>
<td>Keeping the pole of the controller in a certain interval.</td>
</tr>
<tr>
<td>Steady-state error</td>
<td>[136, 137]</td>
<td>Analysing the output equation of the system.</td>
</tr>
<tr>
<td>Robustness</td>
<td>[98, 64, 65, 136, 137]</td>
<td>Analysing the feedback loop transfer function.</td>
</tr>
<tr>
<td>Cost of control</td>
<td>[20, 117]</td>
<td>Analyzing a separate cost function.</td>
</tr>
</tbody>
</table>

A description of how the software qualities (adaptation goals) relate to the analyzed guarantees. Hence, we had to infer this information indirectly from the studies:

- Stability indirectly relates to all software qualities that are subject of adaptation and shows the ability of an adaptation mechanism to converge to the goals. However, guarantees for stability are different for different qualities; e.g. lack of stability for a performance goal may imply fluctuations in the throughput of the software application, while lack of stability for a security goal may imply periods with higher vulnerability of the system.

- Settling time is also related to all qualities to be satisfied by the adaptation and shows the time it takes for an adaptation mechanism to bring measured quality properties close to their goals. It is generally acknowledged that the settling time should not be too small as this would compromise stability/robustness, but not too big as this decreases the quality being satisfied [98, 129]. Notably, 7 out of 11 primary studies discussing settling time guarantees are concerned with performance, in particular response time.

- Similarly, overshooting relates to all software qualities that are subject of adaptation and shows how the measured output exceeds the goal during the transient phase. Guarantees for overshoot have a different interpretation for different qualities, e.g., having overshoots on the system response time leads to violation of performance quality. Avoiding overshooting avoids penalties on the respective software qualities [66].
Table 4.13: F18.1 Experimentally verified guarantees

<table>
<thead>
<tr>
<th>Guarantee</th>
<th>Verified Experimentally</th>
<th>Measured by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability</td>
<td>[3, 27, 129, 127, 128, 125]</td>
<td>Ability of the system to achieve its goals. [129] measures stability as the number of system reconfigurations that occur during adaptation.</td>
</tr>
<tr>
<td>Settling time</td>
<td>[50, 127, 128, 125, 91]</td>
<td>The time required to reach the setpoint after a goal change.</td>
</tr>
<tr>
<td>Overshoot</td>
<td>[127, 128, 125, 91]</td>
<td>Spikes in the system output for different adaptation options.</td>
</tr>
<tr>
<td>Steady-state error</td>
<td>[50, 127, 128, 125]</td>
<td>Oscillations in the response time of the software for different adaptation options.</td>
</tr>
<tr>
<td>Robustness</td>
<td>[10, 91]</td>
<td>Deviations in the system output under disturbances.</td>
</tr>
<tr>
<td>Optimality</td>
<td>[117]</td>
<td>The tasks completed and the resources used by the software for different adaptation options.</td>
</tr>
<tr>
<td>Cost of control</td>
<td>[38, 105, 104, 81]</td>
<td>The the amount of resources consumed by the adaptation mechanism to achieve the goals.</td>
</tr>
</tbody>
</table>

- Steady-state error relates to all software qualities that are subject of adaptation as well. It shows how big is the amplitude of oscillations of measured output around the setpoint during steady state. For example, in [50] the authors calculate the steady-state error as the mean of the absolute error on a response time requirement. The authors conclude that a higher steady-state error decreases performance.

- Robustness relates to reliability in all primary studies that analyze this property. Indeed, the amount of disturbance the system can withstand directly influences its reliability. One approach to analyze this relation is by adding white noise to the system inputs [65]. Having a more robust software can enhance performance by maintaining low latencies, or increase business value by serving more advertisements on web sites [10].

- Optimality is another control property that relates to any type of software quality. Examples in the primary studies are performance [28], security [66], reliability [89], and business value [88]. Lack of optimality implies that there are no
guarantees that the adaptation mechanism achieves the most favorable output for the software quality under consideration.

- Control cost and overhead relates to efficiency in 6 primary studies that analyze these properties. In these studies the authors look at resources that are spent on satisfying the adaptation goals and on performing adaptation actions. In two cases, controller cost affects system performance as well [81, 117].

To conclude, we look at a number of additional correlations between guarantees and other data items. Correlating the main motivations for control-theoretical software adaptation (item F8) with guarantees shows that 7 of 10 primary studies that stated “formal guarantees” as a main motivation also provide formal guarantees. When correlating software qualities to sensors (item F14, see also Figure 4.11), we obtained the following results: the most frequently used sensors for measuring performance – the primary software quality that is subject of adaptation – are of the software utility and system utilization type. Reliability on the other hand is measured by sensors of the software inefficiency type, efficiency is measured by either resource or system utilization, while business value correlates to software utility and resource utilization. Comparing software qualities with controller types (Figure 4.18), we observe that PID is the dominating type of controller used for all qualities considered in software adaptation, except for accuracy, for which MPC controllers are mostly used. On the other hand, performance is handled by all types of controllers that are applied in software adaptation.

RQ4: Control Guarantees

- Research of software adaptation is primarily focused on software qualities and does not exploit the full potential of control theoretical guarantees.

- Software performance, efficiency, and reliability are the most frequently applied adaptation goals. Business value is an emerging quality goal for software adaptation.

- Most of the primary studies do not provide a tradeoff analysis of system qualities or guarantees.

- Robustness, optimality, and cost are commonly analyzed properties, together with classical control-theoretical guarantees like stability and settling time.

- Guarantees for required properties are provided either by means of formal analysis or by collecting empirical evidence.

- The relation between software qualities and control theoretical guarantees remains largely implicit. We inferred that stability, settling time, overshooting, steady-state error and optimality relate to all quality properties, while robustness relates to reliability, control cost and overhead relate to efficiency.
4.8 Discussion

In this Section, we reflect on the results of the survey focusing on two topics: comparison with the results of the surveys of Patikirikorala et al. [126] and Brun et al. [24], and open challenges for future research in control-theoretical software adaptation.

Before we compare the results of our survey with results reported in [126], it is important to emphasis that the scope of the survey of Patikirikorala et al. [126] is different from our survey: while we concentrate on the adaptation of software, in particular application software and supporting middleware services, [126] does not distinguish between control-based adaptation at different layers of computing systems. Furthermore, a large part of the results of our survey cannot be compared since [126] does not consider important aspects of control-based adaptation, including model properties such as model linearity, time framework, model time dependencies, actuators, controller purpose, guarantees, among others items that we collected and analyzed.

Nevertheless, we can compare the following:

1) **Model type.** The ratio between black-box plus grey-box models and analytical models in our survey is similar with the results of [126] (about 65/35). However, a notable distinction concerning model type is that [126] does not distinguish between black box and grey-box models. As shown in our review, the difference is very relevant. In our survey, black-box models are used rarely (3 studies compared to 23 studies that use grey-box models) and, in most cases, black-box models are used as a part of generic frameworks. As for types of analytical models, [126] reports that almost half of the analytical models are queuing network models. Our survey, on the other hand, found that different types of analytical models are used, with only 2 of 11 studies using queuing networks.

2) **Sensors** (referred as “performance variables” in [126]). The most frequently used types of sensors reported in [126] are response time, resource utilization and system utilization, and “hit or miss ratio.” Resource utilization and system utilization directly map to the same sensor types in our survey. Response time and “hit or
miss ratio” fit under sensor types software utility/inefficiency in our survey. However, other sensor variables specific to software adaptation, as listed in Table 5.1 are not reported in [126].

3) Controller type. As [126] classified controllers together with composition schemes, the reported results are hard to compare with the results of our literature review. However, we can still see that PID controllers are the dominant type of controllers that emerged in both surveys. On the other hand, MPC was much more used in primary studies of our survey compared to [126]. Optimal control (see LQR in [126]) and feedforward control were used in a small number of analyzed studies in both surveys. As for the controller composition schemes, both our review and [126] found studies that use hierarchical, cascaded, and switching control. However, the number of such studies was relatively low in both surveys.

4) Controller adaptivity and composition scheme. In our survey, adaptive controllers were used in almost 50% of the primary studies, while [126] reported only 15% for this data item. Explaining such a mismatch is not difficult because [126] classified adaptive controllers in a separate group, without identifying which types of controllers (PID/MPC/etc.) were adaptive.

5) Assessment Approach. The ratio of studies that used example applications and simulation as assessment approach compared to other assessment approaches is approximately equal in both our survey and [126]. As a side note, [126] refers to example application as “case study with a test bed.” According to our observations, almost none of the primary studies applies a scientifically valid case study approach, but rather provide results of one or two adaptation scenarios. Moreover, some of the studies justify their approach only with discussion.

6) Application Domain. Although [126] does not specify the precise application domains (e.g., middleware, data storage, and virtual machine are technologies rather than application domains), the authors noted that many analyzed approaches deal with managing web/application servers. In our review we observed a similar trend with studies from the e-commerce domain, where content was optimized on the server side of the application. It is also notable that RuBIS was one of the most used benchmark in both surveys.

Comparison with Brun et al. [24].

Although the article by Brun et al. [24] is not based on a systematic analysis of the state of the art and has a broader focus as this systematic literature review, we can find a number of commonalities and differences compared to the results of our review.

[24] discusses the role of feedback loops in self-adaptive systems in general and from a control engineering perspective in particular. The authors state that a key reason for using feedback control is to reduce the effects of uncertainty which appear in different forms as disturbances or noise in variables or imperfections in the models of the environment used to design the controller. The main motivations for applying control theory to software adaption derived from the primary studies of our review are formal guarantees, the maturity of the field of control theory, and the effectiveness of control theory. Inline with [24], uncertainty is a basic under-
lying reason for applying self-adaptation, however, our survey provides concrete arguments why authors have applied control theory to realise adaptation.

The part of [24] that focusses on control theory in particular is on adaptive control. The authors discuss Model Identification Adaptive Control (MIAC) and Model Reference Adaptive Control (MRAC) that can be considered as two reference models of how adaptive control can be realised. As explained above, the results of our review show that roughly half of the primary studies apply adaptive control. Rather than providing information about what kind of reference model has been used to realise adaptive control, the review results pinpoint: (i) which types of controllers are used in adaptive control, with PID being the dominant type; and (ii) which adaptation techniques are used, which include updating model parameters or switching the model, updating parameters of the control law or changing the law, and involving human operators to make a decision. Some of these approaches realise structural changes that go beyond adaptive control as in MIAC and MRAC.

[24] does not consider many aspects that we studied in our systematic literature review (which was not the particular aim of [24]). These aspects including the formal guarantees that can be provided by applying control theory to software adaptation, system models and their properties, the types of sensors and actuators used, concrete controller types and purposes, and the link between controller properties and software qualities.

Challenges for Future Research

To conclude, we outline a number of challenges that we identified during data analysis and answering the research questions. We clarified particular challenges for software engineers, for control engineers, and for both.

**System models.** The review results show that researchers prefer to work with simple linear time-invariant discrete models. This contrasts with the inherent complexity and non-linear nature of software stated in most of the primary studies. One challenging aspect of linear time-invariant discrete models is their ineffectiveness when the software application is subject to drastic disturbances (for example a sudden change in available resources, or software components that fail). The common solution to handle such situations as used in the primary studies is changing model and/or the controller parameters online. While this solution has shown great potential in traditional control applications, there is a need for substantial evidence to demonstrate its usefulness for handling adaptation of software applications, which is a particular challenge for software engineers.

Complementary to that, an important challenge for software engineers to apply control-theoretical adaptation is to create a mathematical model of the software. [64] suggests exploring known analytical models used in control theory (such as Markov models and queuing networks) to fill the semantic gap between architecture-based and control-based adaptation of software. Along this line, [10] outlines a general control design methodology for queuing networks. Currently there is little research on using non-linear or continuous models to deal with adaptation of software. It would be interesting to investigate whether such models would work better, however, they are complex to build and require sufficient back-
Table 4.14: Software qualities versus control theoretic guarantees

<table>
<thead>
<tr>
<th>Control Guarantee</th>
<th>Quality Properties</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability</td>
<td>All, indirectly</td>
<td>Guarantees on the ability of the system to converge to the goals. This connection is one-directional, i.e., a system can be stable without goals, but a goal cannot be achieved in an unstable system. Different interpretation for different qualities.</td>
</tr>
<tr>
<td>Settling time</td>
<td>All</td>
<td>Guarantees on time it takes to bring measured quality property close to its goal. Settling time should not be too small (for stability/robustness) but also not be too high (decrease of quality).</td>
</tr>
<tr>
<td>Overshoot</td>
<td>All</td>
<td>Guarantees on the degree the measured output exceeds the goal in transient phase. Different interpretation for different qualities.</td>
</tr>
<tr>
<td>Steady-state error</td>
<td>All</td>
<td>Guarantees on the amplitude of oscillations of measured output around the setpoint during steady state. Different interpretation for different qualities.</td>
</tr>
<tr>
<td>Robustness</td>
<td>Reliability</td>
<td>Guarantees on the amount of disturbance a system can withstand; relates directly to reliability of the system.</td>
</tr>
<tr>
<td>Optimality</td>
<td>All</td>
<td>Guarantees that the system reaches the most favourable output for the given quality.</td>
</tr>
<tr>
<td>Control cost and overhead</td>
<td>Efficiency</td>
<td>Guarantees on the resources used for satisfying goals and performing adaptation actions.</td>
</tr>
</tbody>
</table>

ground in control theory. Consequently, software engineers may involve control engineers when tackling this challenge.

As for the model type, grey-box models were applied in almost 60% of the software systems of the primary studies. As these models reflect only particular parameters of the system, an open question is: how to choose the system parameters to be modeled and what techniques to use in order to identify those parameters? One generally applicable grey-box model was found during this review, see the LLR model in Table 4.8. However, most of the grey/black-box models used in the primary studies were developed to handle a specific case. Both software and control engineers should devote more efforts on identifying generic grey/black-box models for different types of software systems.

**Sensors and actuators.** The review results show that software adaptation requires new types of software sensors, as well as actuators that have a direct effect on the application software. We observed that in systems where actuation time
is critical, the authors suggest taking the cost for adaptation into account when designing the adaptation mechanism. While we were able to provide a broad classification of the types of sensors and actuators used in software adaptation, there is currently no clear view on how sensing and actuating of software for control-theoretical adaptation can be supported in a systematic way, both from an architecture and implementation point of view. Hence, challenging questions both for software and control engineers are: (a) how to translate software qualities, such as security and resilience, to setpoints? (b) what sensors could be used to measure particular software qualities? (c) how to translate controller outputs to actuators that effect the software? (d) how to ensure locality and consistency of adaptation, how to support quiescence for control-theoretical adaptation of software?

**Controllers.** We observed that the choice of particular controllers depends on the problem at hand. For software with a single adaptation goal, adaptive PI controller is the preferred choice in the primary studies. For software with multiple adaptation goals preference is given to MPC and optimal controllers. However, several aspects regarding the choice of controllers remain open for further research. Open questions both for software and control engineers include: Are the current solutions scalable to real-world systems? Or even stronger: for what types of real software systems are controllers applicable? What are appropriate controllers to deal with priorities and tradeoffs among quality goals in software adaptation? What controllers are suitable for handling uncertainties in software systems that can only be resolved at runtime? Can we utilize the reusability and portability techniques from software engineering to design reusable controllers?

**Guarantees for adaptation goals.** Our review shows that control-theoretical software adaptation is concerned with addressing typical software goals, in particular performance, efficiency, and reliability. As modern software systems often need to be designed with partial knowledge, providing guarantees is essential. However, we observe that formal analysis of guarantees is poorly exploited in most of the primary studies. One challenging aspect of software adaptation that we tried to address in this literature review is connecting software qualities to control theoretical guarantees. Table 4.14 summarizes the results. As most authors do not provide an explicit connection between control theoretic guarantees and quality properties, it would be interesting to further investigate this connection with future primary studies. Such study would benefit from joint efforts of software and control engineers. An open challenge that comes from the implicit connection between software qualities to control theoretical properties is to select the proper control techniques in order to satisfy the quality properties specified by the stakeholders.

### 4.9 Threats to validity

To increase the quality and soundness of the review results, we followed a systematic approach. However, we point to possible threats to validity.

**Internal validity:** the extent to which a causal conclusion based on a study is warranted. The topic of this literature review lays at the intersection of two very
different disciplines: control theory and software engineering. The disciplines have a different culture and use different vocabulary. Even the term “adaptive” has a different meaning in these two communities (see clarification in the introduction of the paper). To address this threat, the research team involved in this survey was balanced with an equal number of researchers from both disciplines. The researchers had comparable experience and worked closely together during all phases of the review process. In addition, our particular focus was on software adaptation that uses classical or advanced control techniques. Deciding whether a study should be included or not, was not always straightforward, in particular regarding the adaption of software at application and high-level middleware level, and inclusion of some areas of control theory, such as discrete event control. To mitigate this threat, the decision on study inclusion was always based on agreement between at least two researcher that independently checked the papers. In case of disagreement, a third researcher was consulted and after discussion, a decision was made in consensus.

**External validity:** the extent to which the findings can be generalized to all control-theoretical software adaptation research. We acknowledge that limiting the automatic search to selected venues and applying an automatic search strategy using a selection of search engines, we may have missed some primary studies. To preempt this threat, we took several measures. First, during the selection of the venues we followed a thorough process in which the review team worked closely together and consulted with experts of the two disciplines to crosscheck and identify missed target venues. In this process, we followed an inclusive policy, without compromising on the expected quality of primary studies. Second, we started the search process with pilot searches to define and tune the search string, crosschecked the data using both general-purpose and scientific search engines, actively involved expertise of colleagues in the selection process when needed. Thirdly, we performed snowballing to find potentially missed material.

**Construct validity:** the extent to which we obtained the right measure and whether we defined the right scope in relation to what is considered research on software adaptation. The definition of control-theoretical software adaptation we used in this survey (see Section 4.2) may be biased and the list of extracted data items (Section 4.6) may be incomplete. Regarding the scope on software (application software and high-level middleware services), we relied on well-established insights from the field of software engineering. Regarding the scope of adaptation mechanisms, we acknowledge that there is not a general consensus on what is considered control-based adaptation. Our choice to limit the scope to classic and advanced control theory is motivated by the very different nature of realiseing adaptation with other related paradigms. To address this threat, we consulted with researchers from both software engineering and control theory domains, as well as utilized experience of related surveys, such as [126]. Finally, there may be threat regarding the quality of reporting of studies that may have affected both the selection of papers and the extraction of data. To anticipate this threat, we extracted data about the quality of reporting. We found out that many primary studies reported only results from successful experiments and did not acknowledge threats to validity. Hence, our review may not show particular limitations of
control-theoretical software adaptation. But in general, the reporting quality of the primary studies was good, which provides a basis to make conclusions about the validity of extracted data.

Reliability: extent to which we can ensure that our results are the same if our study would be conducted again. The researchers involved in this survey may have been biased when collecting and analyzing data of studies. To address this threat, the team defined a detailed protocol [135] for the survey that provides an explanation of the survey goals, the data items that are collected, the analysis performed, and the techniques applied to classify results. In particular, data extraction and analysis was done by two researchers in parallel and further discussed in case of differences in opinions to increase confidence. Nevertheless, the background and experience of the researchers may have created some bias, and introduced some level of subjectivity in some cases. This threat is also related to conclusion validity, which is concerned with the ability to replicate the same findings.

4.10 Conclusion

In this paper, we reported the results of a systematic literature review that aimed to shed light on the use of control theory as a paradigm for designing adaptive software. The study results show that control-theoretical software adaptation research is still in a preliminary stage. The number of studies is still low, but we observe a rapid growing interest in the field over the last years. We also found a number of studies where control theory was applied to the software artifacts in the development life cycle, which indicates about the research interest in a broader use of control theory for self-adaptation.

Despite software is usually considered highly non-linear, the majority of the studies use simple linear models. All studies evaluated their work with simple applications or simulations. This raises questions about how well the current approaches, in particular with simple linear models, will scale to real-world applications, or whether other approaches need to be explored. To achieve the quality goals of software applications, these goals have to be translated into control goals (setpoints). Furthermore, to adapt the software and measure the effects of the controller actions, the software applications need to be instrumented with sensors and actuators. There is currently no clear view on how this translation can be done in a systematic manner and how sensors and actuators for control-theoretical software adaptation can be realized in an effective way. Finally, the key driver to explore control-theoretical software adaptation reported in the studies is the formal underpinning of control theory as a basis to provide guarantees for adaptation goals. This survey shows that classic controller guarantees are poorly exploited when engineering control-based solutions. Explicitly linking control theoretic guarantees to software qualities is a challenging topic for future research.

To conclude, we would like to emphasize that research on control-theoretical software adaptation is situated at the crossing of two disciplines: software engineering and control theory. Traditionally, these disciplines operate in different worlds, but progress in these fields requires that both disciplines take an open po-
sition to one another. Without the joint effort of researchers from both disciplines this survey would not have been possible. We hope that the outcome of this joint effort may be a stimulus for new research in this exciting area.
Chapter 5

SimCA: a Control-Based Approach to Adapt Software Systems

In this Chapter, we present a reusable CBSA approach called SimCA (Simplex Control Adaptation) that satisfies S-reqs and O-reqs in the presence of disturbances or measurement inaccuracies.

SimCA provides an initial answer to the research question \( RQ_2 \) “What are the appropriate models and control solutions that can be used to address STO-reqs, and deal with disturbances and goal changes?” and research question \( RQ_3 \) “What software qualities can be satisfied and what guarantees are provided with a CBSA approach that deals with STO-reqs, disturbances and goal changes?” by:

- Introducing a formal model of a self-adaptive software system. In SimCA we formalize software applications from two domains as a linear time-invariant discrete grey box model which automatically updates according to runtime variations.
- Creating an adaptation solutions, namely a combination of multiple PI controllers with the Simplex optimization algorithm, able to adapt the system according to S-reqs and O-reqs in the presence of environmental disturbances and inaccurate measurements.
- Performing a formal analysis and experimental verification of the following guarantees: stability, overshooting, settling time, robustness to disturbances, steady-state error, detection of infeasible solution.
- Through informal exploratory case studies with the UUV system and the TAS exemplar, performing an experimental analysis of qualities, such as performance, reliability, cost, resource consumption, achieved by self-adaptive software equipped with SimCA.

This chapter presents our article published at the Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE 2016) [137].

75
Abstract

An increasingly important concern of software engineers is handling uncertainties at design time, such as environment dynamics that may be difficult to predict or requirements that may change during operation. The idea of self-adaptation is to handle such uncertainties at runtime, when the knowledge becomes available. As more systems with strict requirements require self-adaptation, providing guarantees for adaptation has become a high-priority. Providing such guarantees with traditional architecture-based approaches has shown to be challenging. In response, researchers have studied the application of control theory to realize self-adaptation. However, existing control-theoretic approaches applied to adapt software systems have primarily focused on satisfying only a single adaptation goal at a time, which is often too restrictive for real applications. In this paper, we present Simplex Control Adaptation, SimCA, a new approach to self-adaptation that satisfies multiple goals, while being optimal with respect to an additional goal. SimCA offers robustness to measurement inaccuracy and environmental disturbances, and provides guarantees. We evaluate SimCA for two systems with strict requirements that have to deal with uncertainties: an underwater vehicle system used for oceanic surveillance, and a tele-assistance system for health care support.

5.1 Introduction

The ever growing demand on software has drastically increased the burden on software engineers. Customers expect software to cope with continuously changing conditions. They expect the software to deal seamlessly with varying resources, mask sudden failures, and adapt to changes in system goals. Often, these changing conditions are difficult to predict at design time and handling these uncertainties has become an important concern of software engineers.

Self-adaptation is widely encouraged to address such uncertainties [42, 49]. Self-adaptation handles uncertainties at runtime, when the knowledge becomes available. To that end, the system is equipped with a feedback loop that monitors the system and environment and adapts the system to meet the requirements under changing conditions. As more systems with strict requirements require self-adaptation, providing guarantees for adaptation has become a high-priority concern [142, 35, 41, 155]. Architecture-based approaches for self-adaptation [123, 103, 154], where feedback loops consist of components that realize monitor-analyze-plan-execute (MAPE) functions, have been widely used to
ensure system goals under uncertainty. However, recent research has pointed out that providing assurances for such systems is very challenging [31, 151], calling for new perspectives on engineering self-adaptive systems.

More than a decade ago, Hellerstein et al. [80] argued for using principles from control theory as a solution for runtime adaptation with formal guarantees. This viewpoint has recently gained increasing attention, e.g., [67, 48]. In this approach, a software system is treated as a plant to be controlled, and a control feedback loop empowers the software with self-adaptation capabilities, providing formal guarantees, regardless of uncertain operating conditions [23, 62].

Recently, a strategy for applying control theory to computing systems in a general way has been proposed in the form of the Push-Button Methodology (PBM) [65]. PBM can automatically build a controller of an adaptive software system that rejects environmental disturbances, while providing control-theoretical guarantees for key properties. As the approach is automated, PBM can be used by practitioners with little control-theoretical background. However, PBM deals only with one quantifiable goal at a time, which is often too restrictive for real applications. For example, consider an e-commerce website that should guarantee particular response times for different categories of customers, using available resources, while maximizing profit from advertisements. Another example is a video streaming service that should provide a particular video quality for each class of customers, employing the available computation facilities, while minimizing congestions along the streaming paths to consumers.

In our research, we focus on one relevant adaptation problem that requires satisfying multiple goals while optimizing the solution according to an additional goal, such as the examples given above. A well-known approach to handle such problems is the simplex method [44] and its variations. However, simplex cannot be applied “as is” to realistic software problems as it can not handle the variety of uncertainties and disturbances that are inherent to software systems. Simplex has no mechanism for rejecting disturbances, transient noise, measurements inaccuracies, etc., nor does it guarantee system stability or absence of errors in the system output. Recent work has explored a control-based approach to handle multiple objectives [66], and pointed to its relevance for practice. However, that approach has restrictions regarding the guarantees it can provide and the engineering support it offers. We further elaborate on this in Section 5.5.

In this paper we present a new approach called Simplex Control Adaptation (SimCA) that aims at solving the problem of adaptation for multiple objectives with guarantees. SimCA builds upon PBM and the simplex method, combining strengths of both approaches. SimCA is able to find a system configuration that satisfies multiple goals, reaches optimality with respect to an additional goal, achieves robustness to environmental disturbances and measurement inaccuracy, and provides control-theoretical adaptation guarantees. To that end, SimCA runs on the fly experiments on the software in an automated fashion, builds a set of linear models of the software at runtime, creates a set of tunable controllers that op-

---

1 In control theory terminology, “plant” usually refers to a physical system that is adapted. It is often called the managed system by software engineers.
erate on these models, and combines controller outputs using the simplex method to adapt the system. The controllers of SimCA use Kalman filters to dynamically adapt the linear model in order to cope with disturbances and non-linearities.

The evaluation of SimCA is conducted in two steps. First, we evaluate the control theoretical guarantees provided by the approach, including system stability, settling time, absence of overshoot and steady-state error, solution optimality, robustness, and detection of infeasible solution. Not achieving these guarantees may violate certain software qualities. For example, lack of robustness guarantees may lead to instability under disturbances, violating reliability requirements. A more detailed mapping between guarantees and software qualities is given in Section 5.4. Second, the effectiveness and generality of SimCA is demonstrated on two cases: a UUV (unmanned underwater vehicle) system performing surveillance missions, and a service-based system for health care. These systems are from different domains, but self-adaptation must guarantee that the strict requirements of both systems are achieved at runtime, regardless of the disturbances. In addition, we provide a qualitative comparison of SimCA with the approach presented in [66].

The remainder of the paper is structured as follows. A motivating scenario for SimCA is introduced in Section 5.2. Section 5.3 presents SimCA and explains how to build self-adaptive systems with the approach. The formal evaluation of guarantees provided by SimCA is given in Section 5.4. In Section 5.5, SimCA is empirically evaluated using two cases. Section 5.6 discusses related work. Finally, conclusions and directions for future research are presented in Section 5.7.

5.2 Motivating scenario: UUV System

We describe a UUV system (based on [134]) that we use as one of the cases to evaluate SimCA in Section 5.5 and to illustrate the technical description of SimCA in the next section. UUVs are increasingly used for a wide range of tasks. Here we look at UUVs used for oceanic surveillance, e.g., to monitor pollution of an area. UUVs have to operate in an environment that is subject to restrictions and disturbances: correct sensing may be difficult to achieve, communication may be noisy, etc., requiring a UUV system to be self-adaptive.

Furthermore, there is a need for guarantees as UUVs have strict requirements, i.e., the system should not impact the ocean area, and since vehicles are expensive equipment that should not be lost during missions.

The self-adaptive UUV system in our study that is used to carry out a surveillance and data gathering mission is equipped with 5 on-board sensors that can measure the same attribute of the ocean environment (e.g., water current or salinity). Each sensor performs scans with a certain speed and accuracy, while consuming a certain amount of energy (see Table 5.1). A scan is performed every second.

The UUV system has to satisfy the following requirements:

R1: A segment of surface over a distance of $S$ (100 km) should be examined by the UUV within a given time $t$ (10 hours in the scenario);
5.2 Motivating scenario: UUV System

<table>
<thead>
<tr>
<th>Parameters of sensors of the UUV.</th>
</tr>
</thead>
<tbody>
<tr>
<td>UUV on-board sensor</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Sensor1</td>
</tr>
<tr>
<td>Sensor2</td>
</tr>
<tr>
<td>Sensor3</td>
</tr>
<tr>
<td>Sensor4</td>
</tr>
<tr>
<td>Sensor5</td>
</tr>
</tbody>
</table>

R2: To perform the mission, a given amount of energy $E$ is available (5.4 MJ in the scenario);
R3: Subject to R1 and R2, the accuracy of measurements should be maximized.

To realize the requirements, sensors can be dynamically turned on and off during a mission. We assume that only one sensor is active at a time, however, we use a combination of sensors during each adaptation period. For example, to perform a mission with energy consumption of 135 J/s we may either use Sensor2 100% of the time, or use Sensor1 50% of the time (using 170*0.5=85 J/s) and Sensor4 50% of the time (using 100*0.5=50 J/s).

The requirements R1 and R2 are critical to the success of the surveillance mission, but they may change at runtime due to unpredictable events in the environment. In addition, the adaptation task is not trivial because the system is affected by different disturbances such as:
- Fluctuations in the expected behavior of the UUV (actual scanning speed or energy consumption differs from the specification) up to $\pm 10\%$ of the expected values;
- Inaccurate measurements: e.g., the monitoring mechanism reports a scanning speed of 2.6 m/s instead of the actual value of 2.2 m/s;
- Constant deviations of the sensor output due to a sensor problem, e.g., a sensor starts consuming 50% more energy than stated in the specification;
- Sensor failures;
- Gaussian or Random noise in the communication channel, which may cause errors of the communicated data.

In summary, to realize its mission, the UUV needs to self-adapt to changes in requirements and rejects different types of disturbances. The achievement of goals must be guaranteed.

**Problem definition:**

The general adaptation problem we aim to solve is the following:

*To guarantee the satisfaction of multiple goals and optimize the solution according to another goal, regardless of possible fluctuations in the system parameters, measurement accuracies, requirement changes, and dynamics in the environment that are difficult to predict.*

The UUV scenario offers one concrete instance of this general problem. Defining and developing an adaptive solution for this general problem introduces several key challenges. First, the appropriate adaptation sensors (measured variables) and
actuators (knobs that can influence the software behavior) must be carefully chosen. Second, the software system must be modeled. Third, the appropriate adaptation mechanism that controls the model and satisfies multiple goals, while rejecting external disturbances, must be developed. Fourth, the system must incorporate an optimization approach to optimize the solution according to additional goal. The following section proposes SimCA that aims to address these challenges.

### 5.3 Simplex Control Adaptation

To build an adaptive system with SimCA, the approach requires four elements from a software engineer:

1. A working prototype of the software (plant).
2. A set of quantifiable goals to be controlled plus one optimization goal. For requirements with time-dependent constraints (e.g., a constraint on the available energy to be used in time window), we transfer the constraints to setpoints that satisfy the constraints over time.
3. Tunable parameters (actuators) that can be used to adapt the running system to address the goals.
4. Adaptation sensors\(^1\) to measure the effect of the adaptation on the system.

With these four elements SimCA is able to build a self-adaptive system that solves the adaptation problem formulated in Section 5.2. In order to use SimCA, engineers do not need to construct software models. Instead, the approach works in three runtime phases:

- First, in the *Identification* phase, SimCA synthesizes models that capture the dependency between the adaptation parameters and the measured system outputs.
- Second, in the *Controller Synthesis* phase, SimCA constructs an appropriate set of controllers for the synthesized models.
- Third, in the *Operation* phase, the controllers carry out control and the outcome of multiple controllers is combined using the simplex method to optimally drive the outputs of the system towards the set goals.

The three phases of SimCA are performed during system operation. We describe the phases in detail in the following subsections.

#### UUV scenario

Before that, we illustrate the four elements required from a software engineer to apply SimCA to the UUV system:

1. A working prototype is the UUV system itself.

\(^1\)Not to be confused with UUV sensors in the motivating case.
2. The quantifiable goals are the scanning speed and energy consumption, the optimization goal is the measurement accuracy. We transform requirements R1 and R2 into quantifiable goals as follows: we keep the average scanning speed on a particular level such that the target area of surface is examined in the given amount of time; similarly, we keep the average energy consumption on a particular level such that the mission is performed with the available energy.

3. The actuator of the UUV system is the combination of sensors that are used for performing the mission.

4. The sensor is the monitoring mechanism that measures the scanning speed and energy consumption of the UUV system during the mission.

**Identification Phase**

During the first phase, a set of $n$ linear models of the controlled system is automatically built, where $n$ is the total number of goals excluding the optimization goal. Each model $M_i$, $i \in [1, n]$, is responsible for one goal $s_i$.

Similar to basic PBM, identification starts by systematically feeding sampled values of the goal $s_i$ in the form of a control signal $u_i$ to the plant and measuring their effect on the system output $O_i$ (see Figure 5.1). The vector of control signals $u_i$ used for identification looks as follows:

$$u_i = [\min_i, \min_i + \delta, \min_i + 2\delta, \min_i + 3\delta, \ldots, \max_i]$$

Where $\min_i$ and $\max_i$ are the minimal and maximum achievable values for the $i$-th goal, $\delta$ is the sampling rate. $\delta$ is a tunable parameter chosen by the system engineer; by default $\delta = (\max_i - \min_i) \times 0.05$. A higher sampling rate will provide a more accurate model, but increase the identification time; whether this is required depends on the domain.

During the Identification phase (and the Control Synthesis phase, see below), the control signal $u_i(k)$ is automatically translated (marked Trans. on Figures 5.1 and 5.2) to an actuation signal before feeding it to the plant. A control signal may for example be translated to the change of a parameter setting of the system or the selection of a component or a service. This translation is performed by the simplex method that serves as a straightforward translator of control signals to an actuation signal during the Identification and Control Synthesis phases. Such simplified translation works because at this stage we need an approximate model and not an optimal solution.

$u_i$ is an array of elements, with $u_i(k)$ being the $k$-th element of that array, where $k$ equals to one adaptation period.
After recording all combinations of control signals and resulting system outputs, the dependency between the control signal $u_i(k - 1)$ and its effect on the measured output $O_i(k)$ is captured by the coefficient $\alpha_i$ which is further used to build controller $C_i$. Coefficient $\alpha_i$ is calculated based on linear regression using the APRE tool [114]. As a result, a set of first order linear models is obtained, representing the reaction of the system to control signals for the different goals:

$$O_i(k) = \alpha_i \times u_i(k - 1) \quad (M_i)$$

The model $M_i$ describes the system behavior, but does not take into account small disturbances or sudden failures that typically can occur in practical software systems. For example, this model may not be able to deal with a particular component failure at runtime.

Earlier work has show that $M_i$ is a linear model that is practical and works for a variety of applications [65], and this is confirmed by the two studies presented in Section 5.5. The different cases have shown that to be effective, the model does not need to capture the precise (usually non-linear) relationship between the control signal and the system output. In addition, the synthesized controller has mechanisms (see the following section) that allow to use $M_i$ even for non-linear systems working under disturbances.

Exploring the effect of a range of values of a goal on the system output during Identification may affect the realization of a temporal constraint associated with that goal. This effect was not taken into account in the original PBM [65]. To ensure that the constraint is not violated, the Model building module measures the time $\Delta t_i$ and resources $\Delta R_i$ spent for Identification, subtracts this amount from the available time $t_i$ and resources $R_i$ respectively and automatically adjusts the quantifiable goal $s_i$ accordingly:

$$s_i = \frac{R_i - \Delta R_i}{t_i - \Delta t_i} \quad (5.1)$$

**UUV scenario**

We illustrate the Identification phase for the energy consumption goal of the UUV system. According to Table 5.2, the minimal available energy consumption is $\min_E = 78$ J/s, while the maximum is $\max_E = 170$ J/s. Then, by default, $\delta = (\max_E - \min_E) \times 0.05 = 4.6$. The model identification starts with sending $u_i(0) = \min_E = 78$ J/s to the plant that is automatically translated by simplex to use Sensor 5 all of the time, because Sensor 5 consumes exactly that amount of energy according to specification. The goal of this procedure is to measure and record the actual output energy consumption of the vehicle $O_E(1)$. After that, the plant receives $u_i(1) = \min_E + \delta = 82.6$ J/s, simplex translates it to a corresponding combination of sensors to be used, and the output $O_E(2)$ is measured again. When the values of $O_i$ are measured for all $k$, coefficient $\alpha_i$ is calculated with the APRE tool, resulting in a system model for the energy consumption goal of the UUV system. We observed a typical value of $\alpha_i$ for this model in the range 0.9...1.1. After the Identification, the goal $s_E$ is adjusted accord-
5.3 Simplex Control Adaptation

![Diagram of simplex control adaptation](image)

Figure 5.2: Control Synthesis phase of SimCA.

According to the amount of consumed energy and time (see eq. 5.1), for the UUV case:

\[ s_E = \frac{(5.4 \times 10^6 - 0.2 \times 10^6)}{(10 \times 3600 - 0.5 \times 3600)} = 152 \text{J/s} \]

Controller Synthesis Phase

The second phase of SimCA, labeled **controller synthesis**, consists of two sub-phases: controller building and controller re-building, see Figure 5.2. Controllers are built once, when the system starts, and may be rebuilt during system operation.

**Controller Building**: during this first sub-phase, a set of \( n \) controllers is built using the set of models \( \mathcal{M}_i \), \( i \in [1 \ldots n] \), each controller managing one goal.

A controller \( C_i \) has one tunable parameter, called **pole** denoted with \( p_i \). To maintain stability and avoid oscillations, the pole value should belong to the open interval \((0, 1)\). The pole is chosen by the controller designer and allows to trade-off certain system properties (see discussion in Section 5.4).

As shown in [65], the system output equation, representing the measured output \( O_i(k) \) in response to a unit step\(^3\) setpoint \( s_i \) is defined as follows:\(^4\)

\[ O_i(k) = s_i \times (1 - p_i^k) \]  \hspace{1cm} (5.2)

By using Z-transform — a frequency domain representation of a discrete time control signal — on \((\mathcal{M}_i)\) and (5.2), and by analyzing the system input-output relationships, the following controller equation can be obtained:

\[ u_i(k) = u_i(k-1) + \frac{1 - p_i}{\alpha_i} \times e_i(k-1) \]  \hspace{1cm} (C_i)

The synthesized controller \( C_i \), \( i \in [1, n] \), calculates the control signal \( u_i(k) \) at the current time step \( k \) depending on the previous value of control signal \( u_i(k-1) \), model adjustment coefficient \( \alpha_i \), controller pole \( p_i \) and the error \( e_i(k-1) \), with \( e_i = s_i - O_i \).

**Controller Rebuilding**: during the second sub-phase, the controllers handle inaccuracies in \( \mathcal{M}_i \). To that end, the controllers of SimCA incorporate two additional mechanisms introduced by PBM:

1. Each controller uses a Kalman filter to constantly update the value of \( \alpha \), adapting the linear model at runtime. This mechanism allows to cope with small perturbations that could not be tracked by non-adaptive \( \mathcal{M}_i \) and assures robustness for non-linear behaving systems.

\(^3\)Step in the setpoint of magnitude one – for example, when scanning speed is required to change from 2 to 3 m/s.

\(^4\)\( p_i^k \) is \( p_i \) to the power \( k \).
2. Each controller is equipped with a change point detection mechanism, which allows to react to unexpected critical changes in the system. The mechanism updates the system parameters or, in some scenarios, re-initiates the Identification phase and rebuilds $M_i$. An example of a critical change may be a software component that suddenly becomes unavailable. Although requiring extra computations, the mechanism is quite simple and makes the controller extremely robust.

**UUV scenario**

We illustrate Control Synthesis with examples. Assume that the identification phase has produced a model for the energy consumption goal with $\alpha_E = 1$. If the engineer has set the pole for the controller to $p_E = 0.9$, then the Controller Building phase will synthesize the following controller:

$$u_E(k) = u_1(k - 1) + 0.1 \times e_E(k - 1)$$

(5.3)

If, during system operation the UUV slows down due to unexpected underwater streams in some area, the Kalman filter will change $\alpha_E$ accordingly and trigger controller re-building. If the change point detection mechanism detects a critical change, e.g. some of the UUV sensors fail, a re-identification will be triggered resulting in a new value of $\alpha_E$ which will be updated in the controller equation.

**Operation Phase**

The third phase of SimCA, labeled *operation* also consists of two sub-phases: control and optimization, see Figure 5.3.

**Control:** in the first sub-phase, the set of controllers effectively perform control. Each controller $C_i$ manages one goal $s_i$, rejects disturbances acting on the according output $O_i(k)$, and provides an output signal $u_i(k)$ that is fed to simplex (see Optimization below). The $\alpha_i$ value of the controller can be updated on the fly by the embedded Kalman filter to handle non-linear system behavior (see Controller Rebuilding). The change point detection mechanism can interrupt the controller to deal with invasive changes of the system. SimCA will then restart Identification, followed by Controller Building.

**Optimization:** during the second sub-phase, SimCA collects all control signals $u_i(k)$ and the system parameters $P(k)$, and passes these data to the simplex block. Simplex calculates the actuation signal $u_{sx}$ that drives the system towards an output that satisfies all adaptation goals.

Generally, the simplex method allows to find an optimal solution to a linear problem written in the standard form:

$$\max \{ c^T x \mid Ax \leq b; x \geq 0 \}$$

(5.4)

where $x$ represents the vector of variables (to be determined), $c$ and $b$ are vectors of (known) coefficients, $A$ is a (known) matrix of coefficients, and $(\cdot)^T$ is the matrix transpose [46].

$^5 P(k)$ contains relevant parameters of system components that can be measured.
5.3 Simplex Control Adaptation

SimCA uses a simplex variant with equalities \((Ax = b)\) because we do not want simplex to change the effect of control signals on the output signals. Instead, simplex is responsible for seamless translation of control signals to actuation signals.

In SimCA each equation, except the last one, represents a goal to be satisfied. The last equation ensures that the system selects a valid actuation signal by constraining the values that can be taken by elements of the vector \(x\), e.g. \(x \geq 0\). The control signals \(u_i(k)\) produced during the control phase replace constants \(b\), whereas matrix \(A\) and vector \(c^T\) are substituted with the monitored parameters \(P(k)\) of the system. The goal of simplex is to find a proper actuation signal \(u_{sx}\), i.e., vector \(x\).

For details on how simplex solves the system of equations (5.4) we refer to the linear programming literature [46, 45, 130].

**UUV scenario**

Assume that the energy consumption goal is set to \(s_E = 152\) J/s. We illustrate how the controller calculate the control signal at time \(k = 200\), assuming that the control signal at the previous adaptation step \(u_E(199) = 149\) and the amount of energy consumed by the UUV at the previous adaptation period \(O_E(199) = 150\) J/s. By substituting the according values in (5.3), we get the control signal value:

\[
u_E(200) = 149 + 0.1 \times (152 - 150) = 149.2\]

The controller will send this value to the simplex block.

To illustrate the optimization sub-phase, we rewrite (5.4) as a system of equations using the UUV scenario:

Maximize **Accuracy**:

\[
\max [\text{Acc}_1 \times x_1 + \text{Acc}_2 \times x_2 + \cdots + \text{Acc}_5 \times x_5]
\]

Subject to:

\[
\begin{align*}
E_1 \times x_1 + E_2 \times x_2 + \cdots + E_5 \times x_5 &= u_1 \\
V_1 \times x_1 + V_2 \times x_2 + \cdots + V_5 \times x_5 &= u_2 \\
x_1 + x_2 + \cdots + x_5 &= 1
\end{align*}
\]

(5.5)

Where: \(x_j\) (with \(j \in [1; 5]\)) is the portion of time (in decimals) the sensor \(j\) should be used during system operation; \(\text{Acc}_j\) is the accuracy of sensor \(j\); \(E_j\) is the
energy consumed by sensor \( j \); \( V_j \) is the scanning speed of sensor \( j \) (for the concrete values of \( \text{Acc}_j, E_j, V_j \), see Table 5.1); and \( u_1 \) and \( u_2 \) are control signals received from energy consumption controller and scanning speed controller respectively.

As it can be observed from the comparison of (5.4) and (5.5), the monitored parameters \( P(k) \) of the system are the sensor energy consumption \( E_j \) and the scanning speed \( V_j \) with the active sensor \( j \). Vector \( c^T \) is replaced with accuracies \( \text{Acc}_j \) of sensors. The last equation of (5.5) ensures that at each time instance during the mission one sensor is working. The vector \( x \) represents the portion of time each sensor should be used during system operation.

### 5.4 Evaluation of Guarantees

We start the evaluation of SimCA by formally analyzing the adaptation guarantees provided by the approach.

**Guaranteed Goal Achievement**

The achievement of system goals (except the optimization goal) is guaranteed by the controllers used in SimCA. Specifically, by using controllers we can formally prove the following four system properties: stability, steady-state error, settling time and overshoot. Stability relates to most software qualities that are subject of adaptation and shows the ability of an adaptation mechanism to achieve goal \( s_i \). For example, lack of stability for a security goal implies periods with high vulnerability of the system. If the system has zero steady-state error, its goal \( s_i \) is reached after a certain time \( \bar{K} \) and \( O_i(k) = s_i(k), k \geq \bar{K} \). \( \bar{K} \) is called settling time, and shows the time it takes for an adaptation mechanism to bring measured quality properties close to their goals. Settling time is computed for a step in the setpoint of magnitude one – e.g., demanding the scanning speed to vary from 2 to 3 m/s. Settling time and steady-state error are also related to most software qualities that are subject of adaptation. For example, fast achievement of an energy consumption goal (with low settling time) means spending less resources in a transient state. Avoiding overshoot, that is, the controlled signal does not exceed the goal before reaching its stable area, avoids penalties on the respective software quality. E.g., an overshoot of system response time may violate a service level agreement. Figure 5.4 illustrates these system properties.

![Figure 5.4: Properties guaranteed by the controllers](image-url)
5.4 Evaluation of Guarantees

The control system used in SimCA is designed to be stable and avoid overshoots, since it has only a single pole and its value $p_i$ belongs to the open interval $(0, 1)$.

To evaluate the steady-state error ($\Delta e$) and unit-step settling time ($\bar{K}$) we recall the output equation (5.2). First, we calculate the system output during steady-state, i.e. when $k \to \infty$. As $p \in (0, 1)$, in this case $p_k \to 0$. From (5.2):

$$O_i(k \to \infty) = s_i \times (1 - p^k) = s_i$$

(5.6)

Based on (5.6), the steady-state error equals: $\Delta e = s_i - O_i = 0$.

Theoretically, it will take infinite time for $O_i$ to converge to the exact value of goal $s_i$, i.e. to make $\Delta e$ zero, we need $k \to \infty$. However, the settling time is formally defined as the time $\bar{K}$ in which the measured variable reaches a value very close to the goal (usually it has reached a certain percentage of the goal value – we denote this value with $s^*_i$). Based on this, $O_i$ can be replaced with $(1 - \Delta s_i) \times s_i$, where $\Delta s_i$ is the difference between $s_i$ and $s^*_i$ in percents. From (5.2) we get:

$$(1 - \Delta s_i) \times s_i = s_i \times (1 - p^k) \Rightarrow k = \frac{\ln \Delta s_i}{\ln |p_i|}$$

(5.7)

From this equation it can be concluded that the settling time $\bar{K}$ of every controller $C_i$ depends on the pole $p_i$: higher values of $p_i$ lead to slower output convergence to the goal value. $\Delta s_i$ is a constant chosen by the system engineer. According to [80, p.85], the common value of $\Delta s$ is 0.02 (2%).

As we are using an instance of simplex method with equalities (see Section 5.3), it will not change the effect of control signal $u_i$ on the output signal $O_i$. Hence, simplex will not alter the mentioned above guarantees provided by controllers.

Guaranteed Optimality and Scalability

Simplex guarantees the optimization goal of the obtained solution. The simplex method was proven to always find an optimal solution (if it exists) to a linear problem [46, 45], such as the one formulated in Section 5.3.

The scalability of SimCA is also inherited from simplex. To understand the scalability of simplex, an interested reader may ask about the number of iterations required to solve a problem using this algorithm. Examples shown in [97] require $(2m - 1)$ iterations worst case, with $m$ the number of equations. Such cases would require too much computation. For practical problems, the method usually finds a solution in just a few iterations [44]. The mismatch between theory and practice is not formulated yet, although a number of efforts have been conducted, incl. the use of probabilistic models to synthesize and solve linear programs to calculate the number of required iterations. Additional details are provided in Section 5.6.

Guaranteed Robustness

By robustness we mean the amount of perturbation the system can withstand while remaining in stable state or the amount of inaccurate estimate in the model the system can tolerate. Robustness directly influences system reliability. In line with
the formal assessment of basic PBM [65], conclusions about the system robustness can be derived for SimCA in a similar fashion: the value of the pole $p_i$ allows to trade robustness for settling time $\bar{K}$.

Formally, the amount of disturbance the system can withstand $\Delta(d)$ by using a controller presented in Section 5.3 can be estimated as follows: $0 < \Delta(d) < \frac{2}{1-p_i^*}$. This means that the value of the pole $p_i$ defines how SimCA will react to disturbances. For $p_i = 0.9$, which is used in most of our experiments, the measurement can be inaccurate by a factor of 20, and the controller of SimCA will still adapt the system to follow the goals. In general, higher values of $p_i$ lead to better robustness while lower $p_i$ decreases the settling time.

Detection of Infeasible Solution

The simplex method brings an additional guarantee for the adaptation strategy: it detects infeasible solutions. According to the principles of linear programming, every linear program (including those solved by SimCA) is subject to one of the following [46, 60]: (1) has an optimal solution; (2) has no feasible solution (e.g., setting the scanning speed of a UUV to 5 m/s which is unreachable with any of the sensors); (3) has an unbounded optimal solution, i.e. the objective function value seeks $\infty$ (or $-\infty$), which occurs if variable values can grow indefinitely without violating any constraint.

As SimCA uses only equalities, it cannot produce an unbounded solution. However, when the goal is infeasible, SimCA will converge to the nearest achievable value of the according goal and alert the user that the goal is not reachable. Such clear detection of an infeasible solution offer an advantage with respect to the basic PBM approach, for which it is unclear if a non-zero error appears due to disturbances or due to an unfeasible goal being set for the system.

Boundaries of Guarantees

First of all, the guarantees are achieved on the model; if the system is not capable to identify a sufficiently good model then the controller will not be able to achieve its goals and guarantees. The importance of successful identification is one of the main reasons to perform it at runtime in real operating conditions. However, as practice shows, even with poor testing of corner cases or transient behavior during identification, the model is representative enough to provide the guarantees.

Second, the guarantees on achieving time-dependent requirements depend on correct measuring the time and resources spent during identification and computing the adjustment of the corresponding goal.

Third, the guarantees are provided after controllers are built, meaning that control-theoretical guarantees do not apply during the Identification and Controller Synthesis phases.

Fourth, in the current realization, SimCA cannot provide guarantees when goals are added/removed at runtime or when the system behavior/architecture is invasively changed.

*Details on how to obtain this formula can be found in [65].
5.5 Experimental Evaluation

We empirically evaluate SimCA with two cases. Section 5.5 describes the experimental setting of the UUV case. Section 5.5 shows the software adaptation performed by SimCA when the goals of the system are changed and in response to variations in the sensor behavior at runtime. In Section 5.5 we show the guarantees provided by SimCA with the case study. The scalability of our approach is tested by adding a panel of sensors to the UUV in Section 5.5. The second case with Tele Assistance System is described and evaluated in Section 5.5. In addition, we provide a qualitative comparison of SimCA with the approach presented in [66] in Section 5.5. Finally, Section 5.5 discusses threats to validity. The experiments are performed on a Dell Notebook with 2.7 GHz Core i7 processor, and 16 GB 1600MHz DD3 RAM. All evaluation material is available at the project website.\(^1\)

**Experimental Setting: UUV case**

We use the UUV system described in Section 5.2 as a primary case to evaluate SimCA. The system is implemented in a Java simulation environment that allows to model and study the behavior of software systems. The initial parameters of the sensors are specified in Table 5.1. The actual data that is used by the adaptation mechanism at runtime is subject to a randomly distributed disturbance up to \(\pm 10\%\) of the expected values, simulating fluctuations of actual parameters of sensors (compared to their specification).

Adaptation is performed every 100 surface measurements of the UUV system: \(k = 100\) measurements, and a measurement is performed each second. At each adaptation step the application calculates the average measured value of the \(i\)-th goal (e.g., energy consumption) during the past 100 measurements. Then it calculates the error \(e_i\) as the difference between \(i\)-th setpoint (e.g., target energy consumption) and the measured value of the \(i\)-th goal. The application also monitors the accuracy of surface measurements.

The task of SimCA is to maximize the measurement accuracy by exploiting the available energy and set the scanning speed to examine the required surface in the given time frame. SimCA achieves this task by calculating the value of the *actuation signal*, which represents the portion of time each sensor \(\{S_1, \ldots, S_5\}\) is used during every adaptation period. As an indication of the complexity of the data used in the evaluation: the total number of sensor configurations that can be selected in the UUV scenario is \(5.5 \times 10^6\).

Due to high dynamics and the unpredictable nature of the environment, the controller pole \(p_i\) in SimCA is set to 0.9 which allows to reject errors/disturbances of high magnitude. \(\delta\) is kept at a default value: \(\delta = (\text{max}_i - \text{min}_i) \times 0.05\).

The application collects the UUV data to build performance graphs, which are used to evaluate SimCA in the following sections. The \(x\)-axis of the graphs are time instants \(k\). Thus, the \(y\)-axis shows the average values of the measured feature per 100 surface measurements of the UUV system.

\(^1\)http://homepage.lnu.se/staff/daweaa/simplex.htm
Adaptation Results

Figure 5.5 shows the adaptation results of SimCA on the UUV system configured according to Table 5.1 and requirements set according to UUV scenario (Section 5.2). Adaptation starts with the Identification phase that is clearly visible when \( k \) is between 0 and 20. At time \( k=20 \) the energy consumption setpoint slightly increases based on the energy consumed during identification (see Section 5.3). The Control Synthesis phase, followed by the Optimization phase, starts after the relationship between control signals \( u_i(k) \) and system outputs \( O_i(k) \) is identified (from \( k \) equals 21 onwards). The two upper plots in Figure 5.5 show that during Operation the system is stable, i.e., the measured energy consumption and scanning speed follow their goals. At \( k = 100 \) we change the available energy change from 5.4 to 5.0 MJ, at \( k = 160 \) we change the distance to be scanned from 10 to 10.5 km. The plots show that these changes in requirements lead to corresponding changes in goals and adaptation of the system output.

Figure 5.5 also shows how SimCA reacts to changes in sensor parameters and sensor failures. At \( k = 220 \), the measurement accuracy of sensor \( S_3 \) drastically decreases from 83% to 43%. With such a low accuracy, \( S_3 \) is not a part of the optimal solution anymore and the system selects a better sensor \( S_4 \) at \( k = 221 \), see the “Sensor usage” plot. At \( k = 290 \), \( S_4 \) stops working, which again leads to switching the sensors to the optimal solution, while the measured energy con-
sumption and scanning speed of the UUV remain on the required level. At this point the measurement accuracy decreases from 87% to 77%. It happens because without $S4$, to satisfy all goals, the system is forced to use $S5$, which has lower accuracy.

The experiment ends at $k = 360$, i.e. after 10 hours of time. The total distance scanned is 10.5 km, the amount of consumed energy is 5 MJ. Over a series of 50 experiments, we measured an error of less than 0.01% on these values.

### Adaptation Guarantees

We now confirm the guarantees formally evaluated in Section 5.4 with the UUV case study.

**Guaranteed Goal Achievement.** SimCA’s guarantees for achieving the are confirmed by the data shown on Figure 5.5:

- The system is stable and converges without overshooting, since it is designed to have only a single pole $p_i$ which belongs to the open interval $(0, 1)$;
- According to the system output equation 5.2, the output $O_i$ during steady-state equals $s_i$ which leads to a zero steady-state error: $\Delta e = s_i - O_i = 0$. The absence of a steady-state error can be observed, for example, on the “Scanning Speed” plot when $k > 25$;
- The settling time $\bar{K}$ of every controller $C_i$ depends on the pole $p_i$ and a constant $\Delta s_i$ chosen by the system engineer: $\bar{K} = \frac{\ln \Delta s_i}{\ln p_i}$. According to [80, p.85], the commonly used value of $\Delta s$ is 0.02 (2%). Hence $\bar{K} = \frac{\ln 0.02}{\ln 0.9} = 40$ adaptation steps. This means that changing the scanning speed from 2.7 to 3.1 (step of amplitude 0.4) would take around $\bar{K} = 40 \ast 0.4 = 16$ adaptation steps. This guarantee can be observed on the “Scanning Speed” plot of Figure 5.5 when $k$ is between 160 and 176;

**Guaranteed Robustness.**

The next experiment shows the effects of the controller pole $p_i$ on the tradeoff between system robustness and settling time. For this, we add a random disturbance of amplitude up to $\pm 25\%$ of the expected values to the energy consumption output signal. Figure 5.6 compares the performance of controllers with $p_i = 0.9$ and $p_i = 0.2$ in such conditions. As described in Section 5.4, adaptation with SimCA is influenced by the values of the pole.

First, a smaller pole leads to a shorter settling time. This effect can be observed when the distance requirement is changed at $k = 200$. The system with a smaller pole (right plots) converges to a new operational goal almost immediately, while a system with a higher pole (left plots) needs 16 adaptation steps to converge. Experimentally we determined that due to fast convergence, the total average accuracy of measurements is 0.1% higher for controller with lower pole.

Second, despite the decrease of the settling time, lowering $p_i$ leads to weaker disturbance rejection. This property of adaptation mechanism of SimCA can be observed after $k = 120$. The system with $p_i = 0.2$ unsuccessfully tries to find an optimal solution until $k = 200$. The system with $p_i = 0.9$ continues working as
expected. Hence, the system with lower pole is not reliable under disturbances of high magnitude.

Another benefit of a high $p_i$ value is less oscillation of sensor usage and a smoother accuracy curve (compare according plots on Figure 5.6). This means that a higher pole value leads to a system that is less responsive to variations in parameters but at the same switches less between solutions.

In addition to rejecting noise and measurement inaccuracy, SimCA can reject constant disturbances. E.g., due to an error, a monitor that measures the vehicle energy consumption can constantly decrease the measured value by 15 $J/s$. The plot on the left side of Figure 5.7 shows the behavior in such a scenario. Although monitoring is not working properly, SimCA still adapts the system by defining proper relationship between control signal $u_i(k)$ and system output $O_i(k)$.

Unlike SimCA, the pure simplex method fails at guaranteeing the control-theoretical properties such as disturbance rejection. Hence, the simplex method produces an incorrect output, see the right plot of Figure 5.7.

Detection of Infeasible Solution. Figure 5.8 shows the detection of an infeasible solution. The energy consumption remains at the required level during the entire experiment.
Initially, we set the goal distance to be examined to 10 km. After Identification ($k_i^20$), the system functions normally. At $k = 150$ the total distance to be examined changes to 13 km, hence the output scanning speed grows until reaching its maximum feasible value of $3.2m/s$ at $k = 155$; and the user is notified about the infeasible solution.

At $k = 270$, we change the distance requirement to an unreachable value of 9 km and the scans start with the minimum possible speed among those sustaining energy consumption at the goal, and the user is notified of the infeasible goal.

It is worth mentioning that getting an infeasible solution does not necessarily mean that the concrete goal is entirely unreachable. For example, the scanning speed of $3.6m/s$ can be achieved by using $S2$ and $S5$. However, the energy consumption goal of $150J/s$ will be violated in such scenario as both mentioned sensors has lower energy consumption. Hence, in case of an infeasible solution the system may inform the user about the contradictory goals set for the system.

**Scalability of SimCA**

To demonstrate the scalability of SimCA we extend the UUV case by significantly increasing the number of possible actuation options (combinations of sensors). In particular, we consider now an UUV equipped with two sensor panels, one on the left side and one on the right side. Each panel is provided with 5 on-board sensors that monitor a surface equal to the surface monitored by the single panel in the

![Figure 5.7: SimCA vs Simplex: constant disturbance.](image)

![Figure 5.8: Detection of Infeasible Solution in SimCA.](image)
original case. The panels simultaneously monitor the respective surface, hence, a combination of two sensors (one from each panel) is used at the same time. The sensors have characteristics similar to those in Table 5.1. Due to space constraints, we refer to the project website for a detailed overview of the parameters of sensor combinations (energy consumption, scanning speed, and accuracy). The task of SimCA is to choose among 25 sensor combinations in order to satisfy the following goals:

R1: The underwater vehicle must examine \( S = 210 \ km \) of surface within a period of \( t = 10 \) hours (i.e., the scanning speed = \( S/t = 5.83 \ m/s \)).

R2: The amount of available energy \( E \) is limited to 5.3 MJ (i.e., mission energy consumption = \( E/S = 147 \ J/s \)).

Note that the scanning speed specified for a combination of two sensors is double the value of the vehicle speed as both panels scan surface in parallel.

Figure 5.9 shows results of a scalability scenario with 2 sensor panels working in parallel. The sensor data, as in the previous experiments, is subject to random disturbances of small amplitude. In general, the system shows the same adaptation behavior (convergence to the goal value, adaptation to sensor parameters change, etc.) as in the case of a single sensor panel, e.g. the change of goals at \( k = 100 \) and 160 switches the sensor combination of the optimal solution. The ‘Sensor Combination Used” plot shows that during operation only 6 of the 25 sensor combinations

\[ J/s \]

\[ m/s \]

\[ \% \]

\[ \% \]

\[ \% \]
are used for this scenario. However, note that during Identification (k = 0 to 20) other sensor combinations are tested as well.

As SimCA has the scalability properties of simplex, we can conclude that increasing the number of on-board sensors will not change the adaptation outcomes.

**Evaluation Scenario 2: TAS**

To show the generality of SimCA, we evaluate the approach with a second case: the TAS exemplar [153]. TAS is a service-oriented application that provides remote health support to patients. The main goal of TAS is to track a patient’s vital parameters in order to adapt the drug or drug doses when needed, and take appropriate actions in case of emergency. To satisfy this goal, TAS combines three types of services in a workflow, shown on Figure 5.10.

![TAS workflow](image)

**Figure 5.10: TAS workflow.**

For service-based systems such as TAS, the functionality of each service can be implemented by multiple providers that offer services with different quality properties: reliability, performance, and cost. The system design assumes that these properties can be quantified and measured. E.g., reliability is measured as a percentage of service failures, while performance is measured as the service response time. At runtime, it is possible to pick any of the provided services.

We consider that five service providers offer the Medical Service, three providers offer the Alarm Service and only one provider offers the Drug Service. Table 5.2 shows example properties of available services based on data from [33].

<table>
<thead>
<tr>
<th>Service</th>
<th>Name</th>
<th>Fail.rate, %</th>
<th>Resp.time, time units</th>
<th>Cost, $</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Medical Service 1</td>
<td>0.06</td>
<td>22</td>
<td>9.8</td>
</tr>
<tr>
<td>S2</td>
<td>Medical Service 2</td>
<td>0.1</td>
<td>27</td>
<td>8.9</td>
</tr>
<tr>
<td>S3</td>
<td>Medical Service 3</td>
<td>0.15</td>
<td>31</td>
<td>9.3</td>
</tr>
<tr>
<td>S4</td>
<td>Medical Service 4</td>
<td>0.25</td>
<td>29</td>
<td>7.3</td>
</tr>
<tr>
<td>S5</td>
<td>Medical Service 5</td>
<td>0.05</td>
<td>20</td>
<td>11.9</td>
</tr>
<tr>
<td>AS1</td>
<td>Alarm Service 1</td>
<td>0.3</td>
<td>11</td>
<td>4.1</td>
</tr>
<tr>
<td>AS2</td>
<td>Alarm Service 2</td>
<td>0.4</td>
<td>9</td>
<td>2.5</td>
</tr>
<tr>
<td>AS3</td>
<td>Alarm Service 3</td>
<td>0.08</td>
<td>3</td>
<td>6.8</td>
</tr>
<tr>
<td>D</td>
<td>Drug Service</td>
<td>0.12</td>
<td>1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

| Requirements | min | 30 | 9 |
The properties of the whole TAS system depend on the choice of concrete service providers that process user requests. For example, invoking $S_1$ and $AS_1$ will lead to the failure rate $TAS_{FR} = S_{1FR} + AS_{1FR} = 0.36\%$, while invoking $MS_2$ and $D$ will lead to the failure rate $TAS_{FR} = S_{2FR} + D_{FR} = 0.22\%$.

The system requirements are the following:

**R1.** The average cost for invoking TAS service is set to 9¢

**R2.** The expected average response time is 30 time units

**R3.** Subject to R1 and R2, the failure rate of TAS should be minimized.

Unlike the UUV case, the TAS is expected to run continuously. The requirements R1-R3 and the properties of the services may change at runtime and the system should adapt accordingly. The adaptation task is to decide, for each request with a patient’s vital parameters, which combination of services to select such that the requirements are satisfied.

The TAS case is realized based on the TAS exemplar [153]. The results of SimCA applied to a TAS scenario are shown in Figure 5.11. The adaptation works as intended: system outputs follow the goal changes at $k = 150$ and 250, the.
optimal solution is changed when $S_5$ stops responding at $k = 370$. As services in TAS fail randomly, the optimal value of fail rate oscillates. However, on average (see the purple line on the “Fail Rate” plot) it decreases from ${\approx}0.37\%$ to ${\approx}0.22\%$ when more resources are available to the application at $k = 150$. Note that SimCA manages to keep the failure rate low with a more strict demand of response time from $k = 250$ onwards.

The TAS case confirms the results obtained with the UUV study. It supports the generality of the approach by showing that SimCA is effective in adapting software systems independent of concrete goals or software components that take part in the adaptation.

**Comparison of SimCA with AMOCS**

Recently, [66] proposed an interesting approach for Automated Multi-Objective Control of Self-adaptive software design (AMOCS in short). AMOCS automatically constructs a system of cascaded controllers to deal with multiple goals. Unfortunately, no replication package was available to quantitatively compare AMOCS with SimCA. Therefore, we perform a qualitative comparison based on the reported results.

Compared to SimCA, AMOCS has the advantage that it does not require the extra optimization step with simplex. Furthermore, the approach supports two schemes for ordering goals: user-defined prioritization of goals, and automatic ordering where the controller automatically ranks goals based on available actuators to achieve as many goals as possible.

However, SimCA’s formal guarantees go significantly beyond these of AMOCS reported in [66]:

- SimCA provides guarantees for robustness and steady-state error; AMOCS’ “robustness analysis is system-dependent and unsuitable for an automated control strategy” [66], while steady-state error is not analyzed.
- Simplex provides optimal solutions, while AMOCS “uses systematic or randomized exploration of solution subspace which introduces an approximation of the optimal solution” [66].
- AMOCS cannot provide guarantees for time-dependent goals (such as the energy consumption goal in the UUV case) because it does not take into account resources spent on learning.
- “Short overshoots are expected in AMOCS” [66]; SimCA can avoid overshoots.

SimCA also provides important engineering support not covered by AMOCS:

- SimCA supports trading settling time with robustness (pole placement). In AMOCS, settling time can just be set based on domain characteristics.
- Sampling and model learning is automated in SimCA. AMOCS requires specialized knowledge and extra efforts for this (e.g., quasi-Montecarlo and grid sampling [66]).
• AMOCS “requires the number of knobs to be greater than or equal to the number of goals” [66]. Finding enough knobs may not be trivial or even be artificial (consider for example the TAS case).

In conclusion, SimCA contributes with a novel control-based approach for satisfying multiple goals that significantly improves over AMOCS, both in terms of the guarantees it provides and the engineering support it offers.

**Threats to Validity**

SimCA can handle one class of adaptation problems (satisfying multiple goals, while optimizing one additional goal), but this class of problems applies to a significant number of systems, as illustrated with the cases used in this paper and for example also those used in [66]. Supporting other types of adaptation goals is subject of future work.

We used standard controller guarantees. In Section 5.4, we provide an initial mapping of the controller guarantees to software quality guarantees. However, additional research is required both to refine and extend this mapping and to understand the coverage of the guarantees that can be provided with the standard controller properties. We did not test the impact of $\delta$ on the model quality guarantees in different operating environments; this could be a part of future work.

Regarding the scope of applicability of SimCA. First, the approach is not applicable to systems undergoing drastic changes in their behavior at runtime as continuous re-identification is very costly. Second, SimCA requires that goals can be quantified as a setpoint, which may not be easy for all properties; an example is security. Third, in the experimental setting we have used only some types of disturbances (e.g., sensor failures and noise). Understanding the impact of other disturbance on the adaptation properties of SimCA requires additional evaluation. We want to highlight that in the current state of the research in control-based software adaptation, it is difficult to outline precise criteria that delineate which systems can/cannot be supported by SimCA (and other approaches such as AMOCS). The study and empirical evaluation of new approaches can contribute to build up this knowledge.

We evaluated SimCA in two domains, focusing on adaption for a typical set of stakeholder requirements (resource usage, performance, reliability, cost). While these systems can be considered as representative instances of a significant family of contemporary software systems, additional evaluation is required to validate SimCA for other types of systems.

Finally, we evaluated SimCA for simulated systems. This is inline with the evaluation conducted by others such as [57, 33, 29]. However, further evaluation of SimCA is required to confirm the evaluation results in real deployed systems.

**5.6 Related Work**

The problem of handling multiple goals in self-adaptation is obviously not new. Most of the existing research, including those in *architecture-based* adaptation
and linear programming, solve this problem by introducing an optimization task that trades off the conflicting qualities looking for an optimal solution. These solution uses many different techniques such as preemption [131] to give preference to more time-critical adaptation requirements, utility functions [43, 109, 40] to optimize component/service selection based on weights of QoS criteria, estimates of performance models [76] to select services with optimal response time, linear programming [39] to deal with different operating environments and conflicting QoS requirements, or combine linear programming with local search [11] to find configurations with minimum cost, and hybrid approaches [141] that first decompose end-to-end QoS constraints into local QoS constraints and then perform local selections. Most of these approaches do not provide the broad set of formal guarantees provided by SimCA. Furthermore, the computational costs of most of the proposed solutions grow exponentially with the size of the problem. SimCA can rely on the scalability of simplex as shown in the evaluation.

An advanced example of an architecture-based solution is the QoSMOS framework [27], which also uses the TAS exemplar for evaluation. QoSMOS employs runtime quantitative verification to provide formal guarantees for satisfying multiple QoS goals, while optimizing cost. The control-theoretical guarantees provided by SimCA are out of focus in [33]. However, the main difference is that QoSMOS requires a set of tools that need to be glued together to realize the feedback loop, while with SimCA, the feedback loop is relatively straightforward and derived automatically.

Besides [66], another approach that trades-off different qualities and provides adaptation guarantees is presented in [62] casting a discrete time Markov model for reliability requirements to a dynamic system. The synthesized controller trades reliability for cost by solving an optimization problem. In [136], compared an initial version of SimCA with ActivFORMS [84], a formally founded architecture-based approach for self-adaptation. The evaluation underlines the pros and cons of both approaches in terms of robustness to disturbances and types the guarantees that can be provided.

Simplex is a proven and practical optimization method [46]. Several variants have been developed for specific classes of problems, e.g. [97]. Simplex and its variants do not require an exponentially growing number of iterations when the problem space increases and do not depend on the structure of equations. Today, Simplex remains a very popular optimization method that is used in a wide variety of domains; recent examples are [40] where the method was used to support exploring optimal controller parameters for complex industrial systems, and [68] where simplex was used to support the exploration of the large design space of a cyber-physical system architecture. Another widespread method to solve optimization tasks is called the interior point method. The choice of simplex over the interior point method in SimCA was based on the scope of the problem: the interior point method is faster but only for specific (usually very large) problems [159].

[52] compared control-theoretical and optimization approaches, showing that continuous controller feedback offers higher potential to meet system goals under constantly changing loads, and provides better settling time and less overshooting. Contrary to using either a control-theoretical or an optimization approach, SimCA
integrates the simplex optimization method with a control-theoretic method (enhanced version of PBM) to endow software systems with the self-adaptive capabilities, exploiting the best of both worlds.

## 5.7 Conclusions

In this paper we presented SimCA: a new approach that allows building self-adaptive software systems that satisfy multiple goals, while reaching optimality with respect to an extra goal. In addition, SimCA achieves robustness to environmental disturbances and measurement inaccuracy, and provides guarantees for the adaptation results. The effectiveness of SimCA was formally evaluated and demonstrated on two cases with strict requirements.

SimCA contributes towards the application of formal techniques to adapt the behavior of software systems, which is one key approach for providing guarantees. At the same time, by automatically building a control mechanism that adapts the software, SimCA does not require a strong mathematical background from a designer, which is a key aspect to pave the way for software engineers to use the approach in practice.

In future research, we plan to study the impact of $\delta$ on the model and extend SimCA to handle on the fly adding and removing goals. Our long term goal is to study and develop reusable control-based adaptation solutions that provide assurances for different types of goals.
Chapter 6

SimCA vs ActivFORMS: comparative analysis

In this Chapter, we present results of a comparative evaluation of SimCA and ActivFORMS (Active Formal Models for Self-adaptation) [84] using the TAS exemplar [153]. This study contributes to answering the research questions RQ3 “What software qualities can be satisfied and what guarantees are provided with a CBSA approach that deals with STO-reqs, disturbances and goal changes?” by:

- Comparing software qualities achieved by SimCA and ActivFORMS in multiple scenarios using the TAS exemplar;
- Comparing achieved guarantees formally and experimentally.

According to the study outcomes, both SimCA and ActivFORMS were able to deal with S- and O-reqs in the presence of disturbances. SimCA coped better with runtime changes and achieved lower cost, while ActivFORMS better supported system that require low settling time. As for the guarantees, SimCA’s design allows to formally verify certain system properties (e.g., robustness, stability, etc.), which hold both at design time and runtime, independently of current system parameters. ActivFORMS provides a different type of guarantees, mostly concentrating on functional correctness of the adaptation solutions.

This Chapter is a copy of our article published in Proceedings of the 1st International Workshop on Control Theory for Software Engineering (CTSE) [136].
SimCA vs ActivFORMS: Comparing Control- and Architecture-Based Adaptation on the TAS Exemplar

Abstract

Today customers require software systems to provide particular levels of qualities, while operating under dynamically changing conditions. These requirements can be met with different self-adaptation approaches. Recently, we developed two approaches that are different in nature — control theory-based SimCA and architecture-based ActivFORMS — to endow software systems with self-adaptation, providing guarantees on desired behavior. However, it is unclear which of the two approaches should be used in different adaptation scenarios and how effective they are in comparison to each other. In this paper, we apply SimCA and ActivFORMS to the Tele Assistance System (TAS) exemplar and compare obtained results, demonstrating the difference in achieved qualities and formal guarantees.

6.1 Introduction

The burden on software developers has drastically increased in recent years as customers expect software to cope with continuous change. They expect the software to run seamlessly on different platforms, deal with varying resources, and adapt to changes in system goals. Often, these runtime changes are difficult to predict, requiring software engineers to design the software with incomplete knowledge.

Self-adaptation is widely encouraged to handle software design with incomplete knowledge [42, 49]. A classic approach to realize self-adaptation is architecture-based adaptation [123, 103], where a system maintains an explicit architectural model of itself, reasons about this model, and adapts itself to particular adaptation goals when relevant changes occur. However, achieving guarantees of the system behaviour with architecture-based adaptation is hard [31]. For this reason adaptation mechanisms based on control theory [80, 67] attracted the attention of self-adaptive systems community.

Recently, we developed two approaches for runtime adaptation, one from each of the mentioned fields: ActivFORMS (Active Formal Models for Self-adaptation) [84] an architecture-based approach, and SimCA (Simplex Control Adaptation) [137] that is based on principles from control theory and linear optimization. In this paper, we present a comparative evaluation of SimCA and ActivFORMS on a set of scenarios. To the best of our knowledge, no systematic comparison between approaches for runtime adaptation based on control theory and architecture-based adaptation has been performed so far. We compare obtained

--

1In this paper by self-adaptive system we mean a system equipped with any kind of adaptation/control mechanism which may or may not be adaptive itself.
6.2 Adaptation Scenario: TAS

In this section, we introduce a scenario of TAS used for evaluating SimCA and ActivFORMS. The main goal of TAS is to track a patient’s vital parameters in order to adapt the drug or drug doses when needed, and take appropriate actions in case of emergency. To satisfy this goal, TAS combines three types of services in a workflow, shown on Figure 6.1.

Each incoming request is first processed by the Medical Service. This service receives messages from patients with their vital parameters, analyses the data, and replies with instructions to (1) change the drug or (2) change the drug doses, or (3) invoke an alarm at the First-Aid squad in case of an emergency. When invoked, the Drug Service notifies a local pharmacy to deliver new medication to the patient or change his/her dose of medication. When the Alarm Service is invoked, it dispatches an ambulance to the patient.

For service-based systems such as TAS, the functionality of each service can be implemented by a number of providers that offer services with different quality properties: reliability, performance, and cost. The system design assumes that these properties can be quantified and measured. E.g., reliability is measured as a percentage of service failures, while performance is measured as the service response time. At runtime, it is possible to pick any of the services offered by the providers. The services are considered to be part of the environment because they are not under control of TAS. For example, the failure profile of a concrete
service implementation may change at runtime, due to the changing workloads at
the provider side or unexpected network failures.

We consider that five service providers offer the Medical Service, three
providers offer the Alarm Service and only one provider offers the Drug Service.
Table 6.1 shows example properties of available services based on data from [32].

Table 6.1: Properties of all services used in TAS.

<table>
<thead>
<tr>
<th>Service</th>
<th>Name</th>
<th>Fail.rate, Resp.time, Cost,</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>% ms ¢</td>
</tr>
<tr>
<td>S1</td>
<td>Medical Service</td>
<td>0.06 22 9.8</td>
</tr>
<tr>
<td>S2</td>
<td>Medical Service</td>
<td>0.1 27 8.9</td>
</tr>
<tr>
<td>S3</td>
<td>Medical Service</td>
<td>0.15 31 9.3</td>
</tr>
<tr>
<td>S4</td>
<td>Medical Service</td>
<td>0.25 29 7.3</td>
</tr>
<tr>
<td>S5</td>
<td>Medical Service</td>
<td>0.05 20 11.9</td>
</tr>
<tr>
<td>AS1</td>
<td>Alarm Service</td>
<td>0.3 11 4.1</td>
</tr>
<tr>
<td>AS2</td>
<td>Alarm Service</td>
<td>0.4 9 2.5</td>
</tr>
<tr>
<td>AS3</td>
<td>Alarm Service</td>
<td>0.08 3 6.8</td>
</tr>
<tr>
<td>D</td>
<td>Drug Service</td>
<td>0.12 1 0.1</td>
</tr>
</tbody>
</table>

The system requirements are the following:

R1. The average failure rate should not exceed 0.03 %
R2. The average response time should not exceed 26 ms
R3. Subject to R1 and R2, the cost should be minimized.

The requirements R1-R3 as well as the properties of the services may change at
runtime and the system should adapt accordingly. The adaptation task is to decide,
for each incoming messages with a patient’s vital parameters, which combination
of services to select in order to continuously satisfy the three requirements.

The Adaptation Problem

Generalizing from the concrete TAS scenario, the adaptation problem we are aiming to solve is the following:

Maintain a desired level of quality for multiple goals and optimize the solution according to another goal, regardless of possible fluctuations in the system parameters, measurement accuracies, requirement changes, and dynamics in the environment that are difficult to predict.

Defining and developing such an adaptive solution introduces several key challenges. First, the appropriate sensors (measured variables) and actuators (knobs that can influence the software behavior) must be carefully chosen. Second, the

*The system design assumes that in case of a failure the request is not dropped and can be send to the same or another service provider for re-execution. Hence, the goal failure rate is lower than the rates of individual implementations.
6.3 Studied Approaches

SimCA

The adaptation logic of SimCA consists of multiple SISO controllers that independently compute control signals for each of the goals, and a simplex block that receives the control signals as input and produces an output that is used for adapting the software system. Figure 6.2 schematically shows the primary building blocks of SimCA.

A detailed explanation of SimCA is available in [137]. In short, each system goal (reliability, performance) except cost is represented as a setpoint $s_i(k)$. On every adaptation step $k$ the system outputs $O_i(k)$ are measured. Based on the error $e_i(k) = s_i(k) - O_i(k)$ and a linear model $M_i$ described below, each controller $C_i$ produces a signal $u_i(k)$ which represent the value of each goal that should be reached by the system. The simplex block receives $u_i(k)$ and additional environment/plant parameters (e.g., the invocation cost of external services) as inputs and produces a simplex signal $u_{sx}(k)$ as output. $u_{sx}(k)$ contains the values of system knobs which affect the plant behaviour. For TAS, $u_{sx}(k)$ is a vector containing the probabilities to select each of the available service providers. Disturbances affecting $O_i(k)$ are handled by controller $C_i$ via adjusting the control signal $u_i(k)$.

SimCA works in three phases: identification, control, and optimization.

In the identification phase, $n$ linear models of the controlled system are built. Each model $M_i$, $i \in [1, n]$, is responsible for one goal $s_i$. The identification phase starts by feeding all possible control signal values to the plant. The goal of identification is to determine the influence of control signal $u_i$ on the corresponding system output $O_i$ at every time step $k$. The dependency between $u_i$ and $O_i$ is captured by the coefficient $\alpha_i$ which is further used to build controller $i$. $\alpha_i$ is cal-
culated during identification based on linear regression using the APRE tool [114]. As a result, the following set of linear models is obtained:

\[ O_i(k) = \alpha_i \times u_i(k - 1) \quad (M_i) \]

In \( M_i \) the control signal is a value from the interval \([min_i, max_i]\), where \( max_i \) and \( min_i \) are the maximal and minimal values that can be achieved by TAS for the \( i \)-th goal:

\[ u_i(k) = (max_i - min_i) \times \eta_i(k) + min_i \quad (6.1) \]

\( \eta_i \) is the control coefficient. During the identification phase, \( \eta_i \) changes from 0.0 to 1.0 using an increment of 0.05 on every step \( k \). As a consequence, most feasible values of goal \( i \) are produced as \( u_i \). This allows us to measure all the possible values of the outputs \( O_i(k) \), calculate \( \alpha_i \), and build \( M_i \).

The model \( M_i \) generally describes the system behavior but does not take into account small disturbances or sudden failures that typically occur in software systems. To deal with inaccuracies in \( M_i \), SimCA uses a Kalman filter to adapt the model at runtime, and a change point detection mechanism that allows reacting to critical changes in the system.

An important note concerning the control methodology of SimCA is that the simplex method does not change the value of control signal \( i \). Instead, simplex is responsible for seamless translation of control signals into a proper actuation signal. For example in TAS, if the control signal equals to failure rate = 0.03, simplex finds a combination of services (S1-S9) that assures this failure rate is not exceeded. Hence, simplex is not considered when building a system model and synthesizing controllers that manage this model. Instead, controllers are assumed to affect the plant output via control signals. Every goal is controlled separately during the first two phases of SimCA. This means that during identification and control phases SimCA works in parallel with multiple Single-Input-Single-Output (SISO) controllers, and then combines control outputs with the help of simplex during optimization phase.

In the control phase, a set of \( n \) controllers is synthesized. Each controller \( C_i \) is responsible for the \( i \)-th goal. \( C_i \) calculates the control coefficient \( \eta_i \) at the current time step \( k \) depending on the previous value of control coefficient \( \eta_i(k - 1) \), adjustment coefficient \( \alpha_i \), controller pole \( p_i \) and the error \( e_i \):

\[ \eta_i(k) = \eta_i(k - 1) + \frac{1 - p_i}{\alpha_i} \times e_i(k) \quad (6.2) \]

The controller pole \( p_i \) belongs to the open interval \((0, 1)\) to maintain stability and avoid oscillations. The pole also allows to trade-off robustness to external disturbances with the convergence speed (known as settling time): higher values of \( p_i \) lead to slower convergence [65].

In the optimization phase, SimCA combines the signals \( u_i(k) \) from multiple controllers using the simplex method to optimally drive the measured output of the system towards its desired behavior. Simplex solves the following problem:

\[ \text{Minimize Cost: } \min C = \sum_{j=1}^{p} c_j \cdot x_j \]
subject to:

\[
\begin{align*}
\sum_{i=1}^{n} \sum_{j=1}^{p} a_{ij} \cdot x_j &= u_i \\
x_j &\geq 0, \quad u_i \geq 0, \quad \text{with } i = 1 \ldots n, \quad j = 1 \ldots p
\end{align*}
\]

(6.3)

where:

- \( p \): a number of variables — each variable represents one service provider;
- \( n \): a number of equations — each equation represents a goal to be satisfied;
- \( x_j \): the values of system knobs, i.e. probabilities to select each of the service providers;
- \( u_i \): the control signals;
- \( a_{ij} \) and \( c_j \): the monitored parameters: the failure rate, response time and cost of external services.

SimCA finds the variables of the problem \( x_j \) and passes them to the plant in the form of a simplex signal. A detailed explanation on how simplex solves the system of equations (6.3) can be found in [137], and additional background in [45].

**ActivFORMS**

ActivFORMS is an architecture-based approach for self-adaptation that uses an integrated formal model of the adaptation components of the feedback loop and the knowledge they share [84]. ActivFORMS distinguishes itself from existing architecture-based approaches in three ways:

- The formally verified model of the feedback loop is directly executed by the virtual machine, hence called the active model. This allows to guarantee at runtime the system properties verified at design time. As the active model is directly executed, the approach does not require coding.
- ActivFORMS supports dynamic changes of the active model. A new feedback loop model can be deployed at runtime to meet new or changing goals.
- ActivFORMS supports goal management and model verification at runtime.

ActivFORMS follows a three-layered reference model proposed by Kramer and Magee [103], see Figure 6.3. The bottom layer comprises the managed system\(^1\) that implements the domain-specific functionality. The active model monitors and adapts the managed system through probes and effectors respectively\(^2\). The goal management layer monitors the active model and environment, and deals with adaptation issues that cannot be handled by the current active model; e.g., it dynamically changes the active model to deal with changing system goals.

The active model realizes a MAPE-K (Monitor-Analyze-Plan-Execute-Knowledge) feedback loop [93] that monitors the managed system and adapts it according to the system goals. ActivFORMS supports feedback loops modeled

\(^1\)Managed system is called plant in control theory.
\(^2\)Probe corresponds to sensor in control theory terminology, effector corresponds to actuator.
using networks of timed automata. A timed automaton is a finite-state machine that models a behavior, extended with clock variables.

Recalling the TAS scenario, we used a utility function as a part of the planner component of a MAPE-K formal model to provide the required adaptation, see Figure 6.4. When triggered by a signal from the analysis component, the planner calculates the utility function coefficients and sends a signal to the execution component to update the managed system.

The utility function works as follows: all system goals (reliability, performance) and optimization goal (cost) are represented as $G_i$. During the first adaptation period the cost goal is set to a minimum achievable value: $\text{min}_{\text{cost}}$. On every adaptation step $k$ the system outputs $M_i(k)$ are measured. Knowing the error $e_i(k) = G_i - M_i(k)$, we calculate how big it is in relation to the actual values of the goal $i$:

$$\lambda_i(k) = \frac{e_i(k)}{\max_i - \min_i}$$  \hspace{1cm} (6.4)

For example, if the measured response time $M_{RT}(k)$ is 29.04 on a first adaptation step and other service parameters are equal to the values from Table 6.1, we get:

$$\lambda_{RT}(1) = \frac{29.04 - 26}{35.4 - 21.68} = 0.22$$

![Figure 6.3: The ActivFORMS approach](image6_3.png)

![Figure 6.4: Planning automaton](image6_4.png)
Then we calculate coefficients $c_{fi}(k)$ for the utility function. $c_{fi}(k)$ represents the selection probability for a QoS strategy in the next adaptation period:

$$
c_{fi}(k) = \frac{\lambda_i(k)}{\sum_{j=1}^{n} \lambda_j(k)} \quad (6.5)
$$

Using the same example, we get:

$$
c_{fRT}(1) = \frac{0.22}{0.30 + 0.22 + 0} = 0.43
$$

In the provided example, a coefficient for performance of $c_{fRT} = 0.43$ means that in the next adaptation period the service provider with the lowest difference between $G_{RT}$ and service response time will be chosen 43% of the times.

For offline model-checking we use Uppaal [18], a tool that supports modeling of behaviors and verification of properties of networks of timed automata. For the specification of properties, we use Timed Computation Tree Logic (TCTL). TCTL allows checking individual states of the system state space as well as traces over the state space. The latter allows to verify reachability, safety, and liveness properties.

The ActivFORMS virtual machine can perform the following functions: execute the formal model according to the semantics of timed automata, interact with the managed system and environment through probes and effectors, support online verification of the active model, and update the active model when requested. ActivFORMS provides a set of formal templates to design the MAPE-K elements [47], and abstract Java classes to implement probes and effectors. Probes track the managed system and the environment and transfer data to the Monitor automata of the feedback loop, while Effectors transfer actions generated by the Execution automata to the managed system.

Goal Management comprises a tree-based goal model where nodes have associated MAPE-K models to realize adaptations. Goal management monitors goals via the virtual machine. When a goal violation is detected, the models associated with an alternative goal that matches the changing conditions are used to update the deployed models via the virtual machine. Goal models can be updated at runtime. Here, we do not focus on the goal layer, we refer the interested reader to [84].

### 6.4 Comparative Evaluation

We now evaluate the approaches. Section 6.4 describes the experimental setting. Section 6.4 shows the adaptation behavior of SimCA and ActivFORMS on a basic TAS scenario and in response to different runtime changes. Finally, Section 6.4 discusses guarantees provided by both approaches.

**Experimental Setting**

We use the TAS case described in Section 6.2 to compare the adaptation approaches. The TAS exemplar [153] is realized as a Java application and extended...
with SimCA and ActivFORMS classes. The starting parameters of the services and system requirements are specified in Table 6.1.

Adaptation is performed once per 100 invocations of TAS ($k = 100$ inv.). At each adaptation step the application calculates the average measured value of the $i$-th goal (e.g., measured failure rate) during the past 100 inv. Then it calculates error $e_i$ as the difference between $i$-th setpoint (e.g., target failure rate) and measured value of the $i$-th goal. The application also monitors the cost of serving the incoming requests.

The task of both SimCA and ActivFORMS is to keep the goals at their setpoints and minimize the cost. SimCA achieves this task by calculating the value of the simplex signal, which represents the probability of selecting the services in the list \{S1, S2, ..., S9\}. ActivFORMS achieves this task by prioritizing goals with the help of utility function and updating cost setpoint in case of goal violations. Due to high runtime fluctuations in the values of service parameters, the controller pole $p$ in SimCA is set to 0.98 which allows to reject errors of high magnitude. For the same reason, the cost increment $\Delta C$ in ActivFORMS is set to a low value of $0.1\epsilon$.

The application collects the data of the system and the service implementations to build performance graphs, which are used to compare the adaptation approaches in the following section. The $x$-axis of the graphs are time instants $t$, each instant corresponds to a series of $\Delta t$ inv. of TAS. Thus, the $y$-axis shows the average values of the measured feature per $\Delta t$ inv. of TAS. $\Delta t$ can be changed in the TAS interface.

**Adaptation Results**

The graphs in Figure 6.5 show adaptation results of SimCA and ActivFORMS on TAS configured according to Table 6.1. SimCA starts with an identification phase ($t < 7$). The control phase, which is immediately followed by an optimization phase, starts after the relationship between control signals, simplex signal and system output is identified (from $t$ equal 7 onwards). This phase is stable, all minor spikes on the SimCA graphs are caused by the random nature of failures in TAS, e.g., a failure rate of 3% does not always lead to 3 fails per 100 inv.

In the same scenario ActivFORMS starts with a cost setpoint of 8.5$\epsilon$ which is the average minimal invocation cost of the TAS workflow. In this scenario we assume no prior offline verification of properties so the optimal cost is considered unknown. As a result, both the reliability and performance goals are violated ($t < 20$ on the right graphs). The adaptation slowly increases the cost setpoint which leads to a zero reliability error and a small performance error after $t = 30$.

Comparing TAS reliability achieved by SimCA and ActivFORMS, it is notable that the latter approach requires three times more time to reach a stable state with no goal violation. Comparing performance, ActivFORMS slightly violates the goal even after $t = 75$, but the most notable error can be observed when $t < 30$. So in this case SimCA needs 4 times less time in order to reach a setpoint. As for the picked services, the strategies differ. Though both approaches use S1

\[\text{We use } \epsilon \text{ symbol to represent cost throughout the paper.}\]
approximately equal amount of times ($\approx 60\%$), SimCA prefers a combination of S2, S4 and AS3, while ActivFORMS uses S2, AS1 and AS3.

The Cost graphs (Figure 6.5) seem to be similar, but closer examination shows that the average cost measured during stable state ($t$ between 30 and 75) is $11.26\$ for SimCA vs $11.37\$ for ActivFORMS. This means a saving of $110\$ per 1000 workflow invocations for SimCA compared to ActivFORMS.

To further compare the approaches, we decrease the measurements interval $\Delta t$ from 1000 to 100 workflow invocations, see Figure 6.6. The measured reliability spikes are almost equal on the top plots, however performance and probabilities to select services in ActivFORMS have noticeably higher oscillation amplitude than in SimCA. Such effect is caused by the nature of the algorithm that selects utility function coefficients: the closer the measured value of a goal $i$ gets to the setpoint $i$, the lower is the priority of goal $i$.

It is worth mentioning that simplex operating under disturbances has the property of distributing probabilities for selecting services equally among all avail-
able services [137]. We do not observe this behaviour in the solution with ActivFORMS. With ActivFORMS, the probabilities of selecting some of the services can suddenly change from 0 to 100% in 1000 workflow inv. (e.g., see the probability to select S2 in $30 < t < 40$). Hence, with SimCA, the load on service providers will be relatively smooth over time, while the load with the ActivFORMS solution can change abruptly, requiring the service providers to keep 100% of their resources available all the time.

Both adaptation approaches have a mechanism that allows to trade-off settling time for other properties, so we study how such trade-off will affect the system output, see Figure 6.7.

Lowering pole $p$ in SimCA from 0.98 to 0.7 drastically increases the oscillation amplitude of all outputs. After $t = 40$ the system even triggers re-identification as the measured system output values are too far from their setpoints. This is caused by a combination of low settling time (i.e., the system immediately follows any change in the output) and small adaptation period that leads to high measurement errors.

Raising the cost increment $\Delta C$ in ActivFORMS from 0.1 to 1.0 also increases oscillations of all system outputs. However, their amplitude is much lower, see the ‘Reliability’ and ‘Cost’ plots. The system is also more stable than in SimCA. This experiment allows to conclude that ActivFORMS is a better adaptation solution for a system with settling time priority.

Finally, we show how both approaches react to runtime changes, see Figure 6.8. First of all, having the same initial conditions as in the previous experiments, we set ActivFORMS’ cost setpoint to 11.0¢ instead of 8.5¢ because we already know the approximate optimal cost value. This adjustment results almost immediately in convergence to the desired setpoint values. Hence, by doing a system pre-run or offline validation, the ActivFORMS adaptation time can be greatly decreased.
Coming back to Figure 6.8, there are three major changes happening in the system at runtime:

- **Goal change.** Both goals are changed at $t=22$: failure rate from 0.03 to 0.05%, resp.time from 26 to 28 ms;
- **Abrupt change.** Medical Service 2 breaks at $t=33$;
- **Parameter change.** Response time of Medical Service 1 increases from 22 to 52 ms at $t=55$.

The plots show that both adaptation approaches deal with all types of change: the measured values of both goals follow their setpoints. As in the previous experiments, ActivFORMS requires more time to adapt to the new setpoints.

More importantly, the cost of TAS workflow execution at $t=22$ decreases to 10.2¢ with SimCA vs 10.8¢ with ActivFORMS. When Service 2 shuts down at $t = 33$, it is already barely used by SimCA because there is another combination of services that can produce the same result with lower cost. However, the ActivFORMS solution even increases the utilization of Service 2 when $t$ is between 22 and 33. When the response time of Service 1 increases at $t = 55$, the difference between the workflow invocation cost is around 1¢ per invocation. This scenario shows that SimCA better solves optimization tasks.

**Guarantees**

By using the simplex method, SimCA guarantees optimal cost in TAS without violating the failure rate and response time goals. Simplex has proven to be a practical and fast algorithm for solving this kind of optimization problems [45].
The control-theoretical guarantees provided by SimCA are confirmed by the data shown on the Figures:

- The control system is asymptotically stable and converges without overshooting, since it is designed to have only a single pole $p$ which belongs to the open interval $(0, 1)$;
- According to the system output equation of SimCA [137], the output $O_i$ during steady-state equals $s_i$ which leads to a zero steady-state error: $\Delta e = O_i - s_i = 0$. The absence of a steady-state error can be observed on the performance plot of Figure 6.5 when $t > 15$;
- The settling time $\bar{K}$ of a unit step of every controller $i$ depends on the pole $p_i$ and a constant $\Delta s_i$ chosen by the system engineer: $\bar{K} = \frac{\ln \Delta s_i}{\ln p_i}$. According to [80, p.85], the commonly used value of $\Delta s$ is 0.02 (2%). Hence $\bar{K} = \frac{\ln 0.02}{\ln p_i}$.
6.4 Comparative Evaluation

\[ \ln \frac{0.02}{0.98} = 193.6. \] This means that changing the response time from 26 to 28 (step of amplitude 0.2) would take around \( 193.6 \times 0.2 \approx 40 \) adaptation steps. This guarantee can be observed on the performance plot of Figure 6.8 when \( t \) is between 20 and 24*;

Due to the nature of the simplex method, SimCA will only work when the setpoint of every goal lays between minimal and maximal values of that goal: \( min_i \leq s_i \leq max_i \). Therefore, when having an infeasible goal, a system with SimCA will not converge to the closest feasible value for that goal. Instead, the user will be notified that the task is not solvable and the change point detection mechanism will cause an infinitely loop of identification phases, see Figure 6.9. On the contrary, ActivFORMS will adapt the system output so that it converges to the closest feasible goal value.

![Figure 6.9: SimCA vs ActivFORMS with infeasible goal (\( \Delta t = 1000 \) inv.)](image)

The ActivFORMS approach provides offline and online guarantees on the system behaviour. The offline guarantees come from the verification of the formal model through TCTL properties at the design time with the Uppaal model checker; verified properties are: reachability, safety, liveness, and deadlock freeness [18].

To achieve the guaranteed cost optimality required by TAS, we first implemented the utility function algorithm as a part of the planner component of a MAPE-K loop. Then, we determined the lowest value of cost that does not violate the goal failure rate and response time with the help of formal offline verification of the following safety property:

\[ A[] (gCost == MIN_VALUE) \implies (measuredFR \leq gFR \text{ AND } measuredRT \leq gRT) \]

The system parameters and goals for verification were taken from Table 6.1. The MAPE-K model is also verified to guarantee the correctness of the algorithm’s functionality. E.g., to guarantee that the system model is deadlock free, i.e. it does not have erroneous states and the system does not get stuck in any particular state, Uppaal provides a special formula:

\[ A[] \text{ not deadlock} \]

The following liveness property guarantees that whenever the planning behavior is invoked to create a plan, then eventually a plan is executed.

*As adaptation is performed every 100 workflow invocations and \( \Delta t = 1000 \) invocations, 4 time steps on the graph equals to 40 adaptation steps*
Planning.CreatePlan --> Execution.PlanExecuted

The runtime guarantees of ActivFORMS are provided in two ways. First, the model that was formally verified offline is directly executed by the virtual machine at runtime so these guarantees are preserved at runtime. Second, as offline verification is limited to the input provided by the verification models, ActivFORMS allows to continue verification of properties at runtime and inform the user about violations. We refer the interested reader to [84] for details.

In the studied TAS scenarios we used both types of runtime guarantees provided by ActivFORMS. The direct execution of the verified model allowed to guarantee the correctness of the adaptation mechanism implementation; the runtime verification mechanism of ActivFORMS guaranteed that the minimal cost, previously determined offline, does not lead to violation of the failure rate or response time goals at runtime. To insure the latter, we added the following runtime goals:

G1: A[] (measuredFR <= gFR AND measuredRT <= gRT)  
G2: A[] (measuredFR > 0.9*gFR AND measuredRT > 0.9*gRT)

Goal G1 assures that the failure rate and response time of TAS do not exceed their goals. Goal G2 assures that the values of both goals will not drop lower than 90% of their goals, i.e. the system will not waste resources. In case of a runtime goal violation, the cost setpoint $gCost$ is adapted accordingly: for G1 violation $gCost$ is increased by the cost increment $\Delta C$, for G2 violation $gCost$ is decreased by $\Delta C$.

Changing $\Delta C$ allows trading-off the speed of the system reaction to runtime changes (settling time) for the amplitude of system outputs oscillations around the goal value (accuracy).

### Disturbance Handling

Throughout the experiments discussed in Section 6.4, both approaches handled randomly distributed measurement disturbances caused by the changes of service parameters at runtime. Different to ActivFORMS, SimCA allows to formally evaluate the amount of disturbance the system can withstand.

According to the PBM approach [65] that lays at the core of SimCA, the amount of disturbance the system can withstand $\Delta(d)$ by using a PBM controller can be estimated as follows: $0 < \Delta(d) < \frac{2}{1-p}$. This means that the value of the pole $p$ defines how SimCA will react to disturbances. For $p = 0.98$ that was used in most experiments, the measurement can be inaccurate by a factor of 100, and the controller of SimCA will still adapt the system to follow the goals.

However, as mentioned in Section 6.4, the simplex method will not be able to reach an infeasible goal. Hence, the measurement inaccuracy on values of goal $i$ that SimCA can withstand is: $\min_i \leq \Delta(d) \leq \max_i$. This property of SimCA can be observed on Figure 6.7 after $t = 40$: the reliability output reaches a value that is higher than maximal achievable value for a TAS workflow which causes a re-identification phase.

When it comes to constant disturbances, both approaches successfully perform adaptation. In the scenario shown on Figure 6.10, there is a constant response
6.5 Discussion

Both SimCA and ActivFORMS successfully solved the adaptation problem. The choice between these approaches depends on the particular scenario. When it comes to formal guarantees, both approaches provide the main guarantee required from adaptation, i.e., to minimize the cost without violating performance and reliability goals. SimCA guarantees this by using the simplex method and ActivFORMS verifies the solution optimality offline. However, at runtime SimCA performs better because offline system validation of ActivFORMS relies on particular system properties that may not hold online.

Other formal guarantees provided by the two approaches are different. With SimCA it is possible to formally prove a number of system properties, such as stability, settling time, amount of disturbance the system withstands, etc., which do not depend on the particular system parameters. On a contrary, with ActivFORMS these properties are inherent to the underlying algorithm, such as a utility function that was used in our study, and do not come from the design of the adaptation mechanism. With ActivFORMS it is possible to formally verify these properties of the algorithm on a particular set of system parameters by exploring the whole state space. However, trying to verify the TAS system with all possible combinations of time measurement disturbance of +3 ms (e.g., the sensor shows 29 ms instead of 26 ms). As in the previous experiments with the same pole and cost increment, SimCA converges to the setpoint faster. It is notable that ActivFORMS achieves better cost by slightly violating the performance goal.

**Figure 6.10:** SimCA vs ActivFORMS with a constant disturbance ($\Delta t = 1000$ inv.)
service parameters will lead to an explosion of the state space. This is compensated by runtime verification.

On the other hand, with ActivFORMS the correctness of the implementation of adaptation mechanism can be formally proven, in particular the absence of erroneous states and correct interaction between adaptation components.

### 6.6 Conclusions

In this paper we presented a comparative evaluation of the two different approaches for self-adaptation: SimCA which is based on control theory and linear optimization, and architecture-based ActivFORMS equipped with a utility function. The study was performed using the TAS exemplar. The analysis of adaptation results have shown that both approaches can deal with multiple goals and provide guaranteed optimality with respect to an additional goal. However, SimCA achieves better results in the presence of runtime changes as it does not rely on data verified at design time. At the same time, ActivFORMS is a good choice for systems that require low settling time.

Except optimality, the two adaptation approaches offer different guarantees. The design of SimCA adaptation mechanism, such as the pole value or the system output equation, allows to formally prove the properties of underlying system and guarantee that they will hold at runtime independent of system parameters. ActivFORMS allows formal verification of system properties based on the particular input data. Runtime verification allows complementary verification when the system parameters change. An important feature of ActivFORMS is that it allows to formally guarantee the functional correctness of the implementation of adaptation algorithm.

This work is a first step towards understanding the effectiveness and formal power of different adaptation mechanisms. As a part of future efforts, we want to compare other adaptation approaches, such as QoSMOS [32] to SimCA and ActivFORMS. We are also planning to test the adaptation mechanisms on different types of systems and scenarios.


Chapter 7

SimCA*: a Control-Based Approach to Handle New and Changing Requirements

In this Chapter, we present a reusable CBSA approach called SimCA* that builds upon SimCA (see Chapter 5) and satisfies STO-reqs in the presence of disturbances and goal changes.

SimCA* completes the answer to research question RQ2 “What are the appropriate models and control solutions that can be used to address STO-reqs, and deal with disturbances and goal changes?” and research question RQ3 “What software qualities can be satisfied and what guarantees are provided with a CBSA approach that deals with STO-reqs, disturbances and goal changes?” initially answered in Chapters 5 and 6.

In particular, SimCA*:

- Adjusts the adaptation solutions with a Goal Transformation phase that addresses T-reqs, hence allowing to satisfy all STO-reqs simultaneously in the presence of disturbances.
- Introduces solutions to support changing system requirements by activating/deactivating/adjusting goals.
- Contains an experimental analysis of qualities, such as performance, reliability, cost, resource consumption, achieved by self-adaptive software equipped with SimCA*. The analysis is performed through informal exploratory case studies.
- Includes a formal analysis and/or experimental verification of the following guarantees: stability, overshooting, settling time, robustness to disturbances, steady-state error, scalability, detection of infeasible solution, detection of unbounded solution.

This Chapter presents a research article published at the Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2017) [2].
Handling New and Changing Requirements with Guarantees in Self-Adaptive Systems using SimCA*

Abstract

Self-adaptation provides a principled way to deal with change during operation. As more systems with strict goals require self-adaptation, the need for guarantees in self-adaptive systems is becoming a high-priority concern. Designing adaptive software using principles from control theory has been identified as one of the approaches to provide guarantees. However, current solutions can only handle pre-specified requirements either in the form of setpoint values (S-reqs) or values to be optimized (O-reqs). This paper presents SimCA* that makes two contributions to control-based self-adaptation: (a) it allows the user to specify a third type of requirement that keeps a value above/below a threshold (T-reqs); and (b) it can deal with requirement sets that change at runtime (i.e., requirements can be adjusted, activated, and deactivated on the fly). SimCA* offers robustness to disturbances and provides adaptation guarantees. We evaluate SimCA* for two systems with strict goals from different domains: an underwater vehicle system used for oceanic surveillance, and a tele-assistance system for health care support. The test results demonstrate that SimCA* can deal with the three types of requirements (STO-reqs) operating under various types of dynamics and the set of requirements can be changed on the fly.

7.1 Introduction

Software applications need, more than ever, being able to deal with change [124, 94]. The need for continuous availability of software requires developers to consider change as part of the development process. Software is expected to deal seamlessly with different types of change, such as varying resources, sudden failures, and changes in the operating environment. Often, these changing conditions are difficult to predict at design time, requiring software to execute with incomplete knowledge and face changing requirements during operation [139, 152]. Consequently, software engineers are developing new techniques to handle change at runtime without incurring penalties and downtime, which is commonly referred to as self-adaptation [42, 49, 149].

Today, many of the software systems need to comply with strict goals, hence requiring guarantees for adaptation, such as robustness to disturbances, system stability and others [142, 35, 155]. Control theory has been identified as a promising approach to design adaptation solutions with formal guarantees [23, 79, 48, 166]. However, most of the approaches using control theory to design self-adaptive systems were developed to solve specific problems for a particular domain. We, on
the other hand, are interested in creating a reusable approach that can satisfy different stakeholder requirements and provide adaptation guarantees under changing operating conditions.

A number of reusable approaches have already been proposed, but they are limited to satisfying certain types of stakeholders requirements. For example, the approach described in [65] can satisfy only one requirement at a time, while other approaches such as [66, 137] can satisfy multiple requirements either in the form of setpoint values (S-reqs) or values to be optimized (O-reqs). However, many software systems today need to address a third type of requirement: a threshold requirement that keeps a value above/below a threshold (T-reqs). A typical example is limiting the response time of a web server. Approaches such as described in [95, 120, 98] solve this problem either by optimizing the response time (O-req) or by defining a setpoint for response time that the controller should guarantee (S-req), when the actual requirement is to keep response time lower than a certain threshold. The idea of T-reqs is similar to the recently explored notion of “constraint” from control theory, see for example [9].

Besides a lack of first-class support for T-reqs, existing approaches also provide limited support for changing the set of requirements during operation, which requires on the fly adjusting, activation and deactivation of requirements. Changing requirements are important in practice, e.g., to deal with drastic changes in the system or its environment that may require the system to change from one set of requirements to another.

In this paper, we use control theory to simultaneously deal with S-reqs, T-reqs, and O-reqs (we refer to a combination of these requirements as STO-reqs) and enable the system to change the set of requirements by adjusting/activating/deactivating requirements at runtime. In particular, we solve a typical adaptation problem inherent to systems with strict goals, that is: to satisfy multiple stakeholder requirements (STO-reqs) that may change at runtime, in the presence of environmental disturbances and inaccurate measurements, and provide formal guarantees on the adaptation results.

To deal with this adaptation problem, we developed SimCA* (Simplex Control Adaptation). SimCA* leverages upon an earlier version that we developed, SimCA [137], that can satisfy S/O-reqs, but is not able to solve the adaptation problem discussed above. Hence, SimCA* is an automated control-theoretic approach to build self-adaptive software systems that satisfy multiple, possibly conflicting STO-reqs, achieves robustness to environmental disturbances and measurement inaccuracy, and provides a broad set of control-theoretical adaptation guarantees.

The evaluation of SimCA* is conducted with two cases from different domains: an Unmanned Underwater Vehicle (UUV) system performing surveillance missions, and a service-based system for health care. Both systems have to operate under disturbances and must self-adapt to guarantee the satisfaction of STO-reqs at runtime, as well as to deal with changes in the requirements.

---

1The “*” symbol refers to the ability of the approach to handle different types of requirements that can dynamically change.
The remainder of the paper is structured as follows. Section 7.2 positions SimCA* in the state-of-the-art control-theoretical approaches for building self-adaptive systems that satisfy multiple requirements. Section 7.3 elaborates on the adaptation problem that we address in this paper and illustrates it with an experimental scenario. Section 7.4 presents SimCA*. The formal guarantees provided by SimCA* are evaluated in Section 7.5. In Section 7.6, SimCA* is empirically evaluated using multiple scenarios of two cases. Finally, conclusions and directions for future research are presented in Section 7.7.

7.2 State of the art overview

There is a vast body of research available that applies principles from control theory to adapt computing systems, see e.g. [126, 111]. However, most of the suggested approaches tend to solve specific problems within a certain domain. Creating a generally applicable approach to build self-adaptive software that satisfies different stakeholder requirements and provides adaptation guarantees has been a topic of research for a couple of years [48]. One of the first attempts to create such an approach is the Push-Button Methodology (PBM) [65]. PBM automatically creates a linear model of software and a controller that adapts the software to meet a non-functional requirements specified by stakeholder. The main limitation of basic PBM is that it can only satisfy one requirement at a time. In recent work [66], the authors of PBM proposed a new approach for Automated Multi-Objective Control of Self-adaptive software (AMOCS in short). AMOCS automatically constructs a system of cascaded controllers to deal with multiple S-reqs and an O-req. The approach supports goal prioritization. Despite the advantages, AMOCS has difficulties with addressing O-reqs as the approach may produce sub-optimal solutions [66] and it is lacking some of the guarantees, e.g. the absence of overshooting. Finally, in [137] we introduced an initial version of SimCA, an approach that builds self-adaptive software able to satisfy multiple S-reqs, while being optimal according to a single O-req. The approach makes the system robust to disturbances and provides a broad set of adaptation guarantees.

However, none of the existing automated approaches can simultaneously deal with a typical set of stakeholder requirements (STO-reqs). The main reason is that control theoretic solutions usually work with goals specified as setpoints (S-reqs). Furthermore, existing approaches cannot handle activation and deactivation of requirements during system operation. This may be too restrictive for practical software system that are subject to continuously change. SimCA* on the other hand can (besides S/O-reqs) satisfy T-reqs that are not typical for control theoretical solutions and deal with adjusting/activation/deactivation of requirements during operation.

7.3 Problem Definition

Based on the analysis of the state-of-the-art, we identified the following problem definition:
To guarantee the satisfaction of multiple STO-reqs and deal with adjustment/activation/deactivation of requirements at runtime, regardless of fluctuations in the system parameters, measurement accuracies, and environmental dynamics that are difficult to predict.

Compared to state of the art approaches, two key challenges must be addressed to deal with this problem. First, the solution must incorporate a mechanism to guarantee the satisfaction of T-reqs. This is not trivial as T-reqs are not a typical type of requirement applied in control theory. Second, the solution needs a mechanism to adapt the adaptation logic on the fly in order to address changing requirements, i.e., adjusting/activation/deactivation of requirements. SimCA* described in Section 7.4 addresses these challenges.

**Problem example: UUV System**

We describe a UUV system (based on [134]) that we use as one of the cases to evaluate SimCA* in Section 7.6 and to illustrate the adaptation problem we aim to solve. UUVs are increasingly used for a wide range of tasks. Here we look at UUVs used for oceanic surveillance, e.g., to monitor pollution of a maritime area.

UUVs have to operate in an environment that is subject to restrictions and disturbances: correct sensing may be difficult to achieve, communication may be noisy, etc., requiring a UUV system to be self-adaptive. Furthermore, there is a need for guarantees as UUVs have strict goals, i.e., vehicles are expensive equipment that should work accurately and productively, and they should not impact the ocean area or get lost during missions.

The self-adaptive UUV system in our study that is used to carry out a surveillance and data gathering mission is equipped with 5 on-board sensors that can measure the same attribute of the ocean environment (e.g., water current or salinity). Each sensor performs scans with a certain speed and accuracy, while consuming a certain amount of energy (see Table 7.1). A scan is performed every second.

**Table 7.1: Parameters of sensors of the UUV.**

<table>
<thead>
<tr>
<th>UUV on-board sensor</th>
<th>Energy cons., J/s</th>
<th>Scan Speed, m/s</th>
<th>Accuracy, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensor1</td>
<td>170</td>
<td>2.6</td>
<td>97</td>
</tr>
<tr>
<td>Sensor2</td>
<td>135</td>
<td>3.6</td>
<td>89</td>
</tr>
<tr>
<td>Sensor3</td>
<td>118</td>
<td>2.6</td>
<td>83</td>
</tr>
<tr>
<td>Sensor4</td>
<td>100</td>
<td>3.0</td>
<td>74</td>
</tr>
<tr>
<td>Sensor5</td>
<td>78</td>
<td>3.6</td>
<td>49</td>
</tr>
</tbody>
</table>

In normal operating mode, the UUV system has the following requirements:

- *R1*: A segment of surface over a distance of \( S \geq 100 \) km should be examined within time \( t = 10 \) hours;
- *R2*: To perform the mission, a given amount of energy \( E = 5.4 \) MJ is available;
• $R3$: Subject to $R1$ and $R2$, the accuracy of measurements should be maximized.

$R1$ is a T-req, $R2$ is a S-req, while $R3$ is an O-req. We use $M1$ to refer to normal operating mode, i.e. the UUV must satisfy the three requirements simultaneously: $M1 = \{R1, R2, R3\}$. In other words, in normal operation mode, the UUV should examine as much surface as possible using all the available energy, while ensuring maximum accuracy. To grasp the difference between T-reqs and S-reqs, if the task of the UUV would be to scan exactly $S$ km in $t$ hours ($R1$) using as little energy as possible of the available $E = 5.4$ MJ ($R2$), while ensuring maximum accuracy ($R3$), $R1$ would become an S-req and $R2$ a T-req. To realize the requirements, sensors can be dynamically turned on and off during a mission. We assume that only one sensor is active at a time, however, we use a combination of sensors during each adaptation period.

In addition to normal operating model, the adaptation should also deal with two additional operating modes. First, when a UUV experiences a sudden energy leak, it must switch to a new operating mode that minimizes the vehicle energy consumption instead of maximizing the measurement accuracy. In this case the UUV will switch to a mode $M2 = \{R1, R2^*\}$, where $R2$ (S-req) changes to $R2^*$ (O-req) defined as:

• $R2^*$: The energy consumption should be minimized;

Second, the vehicle may enter a deep water zone, where sensors fail to produce accurate measurements with a certain rate. The UUV should then switch to a mode $M3 = \{R1, R2, R3, R4\}$, where a requirement $R4$ needs to be activated at runtime, defined as:

• $R4$: The average failure rate should be $F \leq 0.02\%$;

Switching modes and changing requirements is critical to the success of the surveillance mission. The adaptation task is not trivial because the system is affected by different disturbances such as fluctuations in the expected behavior of the UUV sensors, inaccurate measurements, sensor problems and failures, and noise in the communication channel.

In summary, to realize its mission, the UUV needs to self-adapt in order to continuously address STO-reqs, cope with different operating modes by activating/deactivating requirements at runtime, and reject different types of disturbances. These requirements needs to be guaranteed.

### 7.4 Simplex Control Adaptation* - SimCA*

In this section we provide an overview of SimCA*. In particular, Section 7.4 lists the element required for SimCA* to build a self-adaptive system, Sections 7.4 - 7.4 describe different phases of SimCA*, and Section 7.4 shows how SimCA* deals with changing the set of requirements.
7.4 Simplex Control Adaptation* - SimCA*

Required Elements to Apply SimCA*

To build an adaptive system with SimCA*, the approach requires the following elements from a software engineer:

1. A working prototype of the software.
2. A set of STO-reqs to be satisfied.
3. Tunable parameters (actuators) that can be used to adapt the running system to address the requirements.
4. Adaptation sensors to measure the effect of the adaptation on the system.
5. Monitoring mechanisms that notify the system about changing conditions that lead to requirement changes.

As for the second element, it should be possible to quantify the requirements, i.e., transform the system requirements (STO-reqs) into corresponding quantifiable goals (STO-goals). For some requirements this transformation is straightforward (e.g., a requirement of keeping average response time at 3 ms is transformed to an S-goal = 3 ms), while for other requirements with time-dependent constraints the quantifiable goals need to satisfy these requirements over time (e.g., a requirement of using 120 units of a resource in 60 sec is transformed into an S-goal = 2 units/sec).

With the elements listed above SimCA* is able to build a self-adaptive system that solves the adaptation problem formulated in Section 7.3.1 SimCA* works in four phases that are performed during system operation, see Figure 7.1.

- First, in the Identification phase, SimCA uses systematically sampled values of S- and T-goals to synthesize models that capture the dependency between different actuator values (in form of control signals that effect software) and the measured system outputs for these goals.
- Second, in the Controller Synthesis phase, SimCA constructs an appropriate set of controllers for the synthesized models, where each controller is responsible for one S- or T-goal.
- Third, in the Goal Transformation phase, the T-goals are transformed into controller goals (C-goals) using simplex. For a T-goal that needs to keep a value below a threshold the C-goal represents the lowest possible value that satisfies all other goals, for a T-goal that needs to keep a value above a threshold the C-goal represents the highest possible value that satisfies other goals.
- Fourth, in the Operation phase, the controllers carry out control using S- and C-goals as the values to be achieved by the system. Then, the outcome of multiple controllers is combined using the simplex method that takes into account the O-goals and optimally drives the outputs of the system towards the set goals.

---

1 We discuss the scope of applicability of SimCA* in Section 7.6.
SimCA*: a Control-Based Approach to Handle New and Changing Requirements

Compared to the initial version of SimCA [137], SimCA* includes a new Goal Transformation phase and the necessary mechanisms to support changing system requirements by activating/deactivating goals. We start with explaining how SimCA* deals with STO-reqs. Then we explain how the approach deals with changing requirements.

**Figure 7.1: Phases of SimCA***

**Phase I. Identification**

During the Identification Phase, SimCA* synthesizes a set of linear models that capture the dependency between different actuator values (in form of control signals that effect software) and the measured system outputs [137]. Each model $M_i$ is responsible for one S- or T-goal, referred to as $s_i$. As optimization tasks are not solved in the first phases of SimCA*, we take into account only the threshold values of T-goals (and not the values above/below the threshold) during Identification and Control Synthesis. Model $M_i$ is built by systematically feeding sampled values of goal $s_i$ in the form of a control signal $u_i$ to the system and measuring its effect on the output $O_i$:

$$O_i(k) = \alpha_i \cdot u_i(k - 1) \quad (M_i)$$

Coefficient $\alpha_i$ captures the dependency between the control signal $u_i$ at the previous time instance $k - 1$ and its effect on the measured output $O_i$ of S- or T-goal $s_i$ at the current time $k$. The time instances between measurements during identification can be chosen by the system engineer, hence influencing the model quality [137].

Model $M_i$ describes the system behavior ignoring small disturbances and sudden system changes. As small disturbances are difficult to predict at design time and therefore cannot be factored in the model construction, they will be dealt with by using feedback from the running system. Sudden changes will re-trigger the identification phase when necessary.

As exploring different actuator values during Identification may violate requirements with time-dependent constraints, the system measures the time $\Delta t_i$ and resources $\Delta R_i$ (e.g., energy) spent for Identification, subtracts this amount from the available time $t_i$ and resources $R_i$ respectively and automatically adjusts the goal $s_i$ accordingly:

$$s_i = \frac{R_i - \Delta R_i}{t_i - \Delta t_i} \quad (7.1)$$

A similar update of $s_i$ is applied after any goal is changed (where $\Delta t_i$ and $\Delta R_i$ represent time and resources spent before the goal change) as the system under control either over/under-consumes resources during the transition to a new goal.
7.4 Simplex Control Adaptation – SimCA* 

Phase II. Controller Synthesis

In the Controller Synthesis Phase, SimCA* constructs a set of controllers for the synthesized models; each controller \( C_i \) is responsible for one S- or T-goal \( (s_i) \). A controller \( C_i \) has one tunable parameter, called pole denoted with \( p_i \). The pole is chosen by the system designer and allows to trade-off controller responsiveness to change and the amount of disturbance it can withstand [137].

In SimCA*, we use the following controller:

\[
    u_i(k) = u_i(k - 1) + \frac{1 - p_i}{\alpha_i} \cdot e_i(k - 1)
\]

The synthesized controller \( C_i \) calculates the control signal \( u_i(k) \) at the current time step \( k \) depending on the previous value of control signal \( u_i(k - 1) \), model adjustment coefficient \( \alpha_i \), controller pole \( p_i \) and error \( e_i(k - 1) \), with \( e_i \) being the difference between S- or T-goal \( s_i \) and the measured output \( O_i \).

The controller \( C_i \) also handles inaccuracies in the model \( M_i \). To that end, each controller incorporates: (1) a Kalman filter adapting the linear model at runtime; (2) a change point detection mechanism, which allows to react to unexpected critical changes in the system by triggering re-Identification [137].

Phase III. Goal Transformation

The Goal Transformation Phase transforms all T-goals into Controller goals (C-goals), see Figure 7.2; a C-goal represents a particular value of a corresponding T-goal. For example, a T-goal that should keep a value below a threshold will be transformed into a C-goal with a value that is equal to the lowest possible value of the goal below that threshold that satisfies all other requirements. As the values of the C-goals depend on other requirements and system parameters, we use simplex during Goal Transformation. This phase is skipped if there are no T-goals in the system.

![Figure 7.2: Goal Transformation phase of SimCA*](image)

Generally, the simplex method allows finding an optimal solution to a linear problem written in the standard form:

\[
    \max \{ c^T x \mid Ax \leq b; x \geq 0 \}
\]  

(7.2)

where \( x \) represents the vector of variables (to be determined), \( c \) and \( b \) are vectors of (known) coefficients, \( A \) is a (known) matrix of coefficients, and \( (\cdot)^T \) is the matrix transpose [46].
In the Goal Transformation phase of SimCA* each equation, except the last one, represents an S-goal or T-goal to be satisfied. Equalities are used for S-goals, while inequalities are used for T-goals. The last equation ensures that the system selects a valid solution by constraining the values that can be taken by elements of the vector \( x \), e.g. \( x \geq 0 \). The values of S-/T-goals to be achieved replace constants \( b \), whereas matrix \( A \) and vector \( c^T \) are substituted with the monitored parameters \( \mathcal{P}(k) \) of the system (i.e., relevant parameters of system components that can be measured\(^2\)). Note that vector \( c^T \) is replaced with parameters of the O-goals. The goal of simplex is to find a proper combination of variables (vector \( x \)) that satisfies all STO-goals. For details on how simplex solves the system of equations (7.2) we refer to the linear programming literature [46, 45, 130].

Knowing the vector \( x \), each T-goal is transformed into C-goal \( c_i \) as follows: \( c_i = \mathcal{P}_i(k) \times x \). Note that controllers are not involved during the Goal Transformation phase and as such simplex will not change the control signals \( u_i(k) \).

**Phase IV. Operation**

In the Operation Phase, the set of controllers effectively perform control and the outcome of multiple controllers is combined using the simplex method to optimally drive the outputs of the system towards the set goals, see Figure 7.3. A simplex is dealing with the O-goals, only C-goals obtained during Goal Transformation and original S-goals are used in the Operation Phase.

In particular, SimCA* collects all control signals \( u_i(k) \) and the system parameters \( \mathcal{P}(k) \) and passes these data to the simplex block. Similarly to the Goal Transformation phase, SimCA* solves the system of equations (7.2) in order to find a solution (actuation signal \( u_{sx} \)) that drives the system towards an output that satisfies all STO-goals. However, the system of equations (7.2) has now a slightly different structure. First of all, each equation, except the last one, now represents an S-goal or a C-goal to be satisfied. Then, only equalities are used to assure a seamless translation of control signals \( u_i(k) \) to an actuation signal \( u_{sx} \), which allows to sustain all the guarantees provided by controllers. Finally, the constants \( b \) in (7.2) are replaced by control signals \( u_i(k) \) obtained from \( C_i \), which allows to use all the advantages provided by controllers.

\(^2\)E.g. in the UUV scenario \( \mathcal{P}(k) \) are the sensor parameters from Table 7.1
Dealing with New Requirements in SimCA*

We now describe how SimCA* adapts the system when requirements are changed during system operation. To that end, we extended the initial SimCA* workflow shown in Figure 7.1 with additional components, see Figure 7.4.

![Figure 7.4: Dealing with requirements changes in SimCA*. Numbers in circles/diamonds show the sequence of actions.](image)

Any change of system requirements during operation is monitored by the Requirement Monitor that triggers the corresponding adaptation components. Subsequently, we look at the activation of requirements, deactivation, and changing requirement types. Updates of requirements (goal updates) are already explained in the Identification Phase in Section 7.4.

To deal with requirement activation, the Goal Activator first transforms the requirement into a quantifiable goal (see Section 7.4) and reads the relevant parameters $P$ related to that goal. The next actions depend on the type of the requirement that is activated. If an O-req is activated, the Goal Activator inserts $P$ into the objective function $c^T$ of simplex, performs a Goal Transformation (Section 7.4) and proceeds to standard Operation. If an S-/T-req is activated, the Goal Activator performs an Identification for the new S-/T-goal. An advantage of SimCA* is that it does not require a complete re-identification of all goals when a requirement is activated, because each corresponding goal is managed by a separate model-controller pair. After Identification, SimCA* builds a controller for the new goal using Controller Synthesis, followed by a Goal Transformation, after which the system returns to standard operation.

For a requirement deactivation, the Goal Deactivator removes the according elements of the adaptation mechanism. Namely, when an S- or T-req is deactivated, the corresponding controller is removed together with the equation responsible for the goal being deactivated. When an O-req is deactivated, the corresponding variables are removed from the objective function $c^T$ of simplex. Finally, the Goal Deactivator always triggers a Goal Transformation adapting the configuration of the control system to the new set of requirements, after which the system returns to standard operation.

SimCA* also supports changes of requirement types at runtime. To that end, SimCA* performs the following: (i) if an S-req is changed to a T-req (or vice

---

3For example, if the fail rate goal of a UUV ($R4$) is activated, the Goal Activator reads failure rates of all UUV sensors from the specification.
versa), the corresponding equality is changed to inequality in the system of equations (7.2), followed by a Goal Transformation; (ii) if an S-/T-req is changed to an O-req, the parameters \( P \) relevant to this goal are copied from the corresponding equation into the objective function \( c^T \) of simplex. After that, the S-/T-req is deactivated according to the standard requirement deactivation procedure (see above); (iii) if an O-req is changed to an S-/T-req, the O-req is deactivated according to the standard requirement deactivation procedure, while the new S-/T-req is activated according to the requirement activation procedure (see above).

### 7.5 Formal Evaluation of Guarantees

As in the original SimCA, a feature of SimCA* is that it provides a broad set of adaptation guarantees. The guarantees provided by controllers include, see Figure 7.5:

- **Stability**: the ability of an adaptation mechanism to converge to S- or C-goals \((s_i/c_i)\);
- **Absence of overshoot**: the measured quality property does not exceed the goal \( s_i/c_i \) before reaching its stable area;
- **Zero steady-state error**: the measured quality property does not oscillate around goal \( s_i/c_i \) during steady state;
- **Tuneable settling time**: time it takes to bring a measured quality property close to its goal \( s_i/c_i \);
- **Tuneable robustness**: the amount of perturbation the system can withstand while remaining in stable state.

![Figure 7.5: Properties guaranteed by the controllers in SimCA*](image)

Since simplex does not introduce any additional dynamics and works as a straight-forward translator of control signals into an actuation signal, we can formally analyze the following guarantees. The control system used in SimCA* is designed to be stable and avoid overshoots, since it has only a single pole and its value \( p_i \) belongs to the open interval \((0, 1)\). To evaluate the steady-state error \((\Delta e)\), we recall the output equation of control system used in SimCA* [137]:

\[
O_i(k) = s_i \cdot (1 - p_i^k)
\]  

(7.3)
During steady-state the time $k \to \infty$. As $p \in (0, 1)$, in this case $p^k \to 0$. Then, the steady-state error $\Delta e$ is:

$$O_i(k \to \infty) = s_i \cdot (1 - p^k) = s_i; \quad \Delta e = s_i - O_i = 0$$

As for the settling time ($\bar{K}$) and robustness $\Delta(d)$, in [137] we show that by analyzing the control system, we get:

$$\bar{K} = \frac{\ln \Delta s_i}{\ln |p_i|} \quad 0 < \Delta(d) < \frac{2}{1 - p_i}$$ (7.4)

In other words, lowering $p_i$ leads to weaker disturbance rejection but faster responsiveness to change. Note that in the equations above $s_i$ can be replaced with $c_i$ without any effect on the guarantees as C-goals represent particular values (set-points) to be achieved by the system similar as S-goals.

As inequalities are used only to transform T-goals to C-goals, while during operation we are only using the simplex method with equalities (see Section 7.4), it will not change the effect of control signal $u_i$ on the output signal $O_i$. Hence, simplex will not introduce additional system dynamics and will not alter the guarantees mentioned above provided by the controllers. As for the change of requirements, we assume that it will not lead to an unfeasible solution. Under this assumption the guarantees will hold, because changing the number of controllers will not alter the structure of the control system.

The guarantees provided by controllers relate to the quality properties that are subject of adaptation. For example, overshooting on the energy consumption goal leads to an overconsumption of energy (more details are available in [137]).

Simplex provides the following guarantees:

- **Optimality**: achievement of O-goals without violating any of the S- or C-goals. Simplex was proven to always find an optimal solution to systems of equations used by SimCA*, such as the one presented in Section 7.4 [46, 45].

- **Scalability**: small amount of extra time and effort required to solve problems of growing scale. For practical problems, simplex usually finds a solution in just a few iterations [44]. This also ensures that the overhead is low for requirement changes as only one extra simplex iteration is required.

- **Detection of infeasible solution**: ability to detect that the goal $s_i/c_i$ is unreachable. When $s_i/c_i$ is infeasible, SimCA* will converge to the nearest achievable value of $s_i/c_i$ and alert the user.

- **Detection of unbounded solution**: the ability to detect that the objective function value seeks $\infty$ (or $-\infty$). Unbounded solution occurs if values of $u_{sx}$ in simplex can grow indefinitely without violating any constraint, i.e., when the system has contradicting requirements. SimCA* will alert the user about unbounded solutions.

**Boundaries of Guarantees.** First, the **guarantees are achieved on the model**; if the system is not capable to identify a sufficiently good model then the controller will not be able to achieve its goals and guarantees. The importance of successful
identification is one of the main reasons to perform it at runtime in real operating conditions. However, as practice shows, even with poor testing of corner cases or transient behavior during identification, the model is usually representative enough to provide the guarantees. Second, the guarantees are achieved under certain assumptions, e.g. the activation of requirement should not lead to an unfeasible solution (see discussion above). Third, the guarantees on time-dependent requirements depend on correct measuring the time and resources spent during identification and computing the adjustment of the corresponding goal. Fourth, the guarantees are provided after the controllers are built, i.e., control-theoretical guarantees apply only during the Operation phase. Fifth, SimCA* guarantees the STO-reqs regardless of possible dependencies between the goals, to the extent that the goals are feasible (otherwise, SimCA* will alert the user). Finally, in the current realization, SimCA* cannot provide guarantees when the system behavior/architecture is invasively changed.

7.6 Experimental Evaluation

We empirically evaluate SimCA* with two cases. Section 7.6 describes the experimental setting of the UUV case. Section 7.6 shows the software adaptation performed by SimCA* with STO-reqs in different operating conditions. In Section 7.6 we experimentally verify the guarantees and quality trade-offs provided by SimCA*. Section 7.6 describes adaptation in an energy leak scenario when the amount and types of requirements change at runtime. In Section 7.6 we perform experiments with a UUV when a new requirement is activated at runtime. The scalability of our approach is tested by adding a panel of sensors to the UUV in Section 7.6. The second case with Tele Assistance System is described and evaluated in the Appendix. Finally, Section 7.6 discusses threats to validity. The experiments are performed on a Dell machine with a 2.7 GHz Core i7 processor and 16 GB 1600MHz DD3 RAM. All evaluation material is available at the project website.1

Experimental Setting UUV Case

We use the UUV system described in Section 7.3 as a primary case to evaluate SimCA*. The system is implemented in a Java simulation environment that allows to model and study the behavior of software systems. The initial parameters of the sensors are specified in Table 7.1. The actual data that is used by the adaptation mechanism at runtime is subject to a randomly distributed disturbance up to ±10% of the expected values, simulating fluctuations of actual parameters of sensors (compared to their specification).

Adaptation is performed every 100 surface measurements of the UUV system: \( k = 100 \) measurements, and a measurement is performed each second. At each adaptation step the application calculates the average measured value of the \( i \)-th

goal (e.g., energy consumption) during the past 100 measurements. Then it calculates the error $e_i$ as the difference between $i$-th setpoint (e.g., target energy consumption) and the measured value of the $i$-th goal. The application also monitors the accuracy of surface measurements and changes of system requirements. The task of SimCA* is to exploit the available energy and set an appropriate scanning speed in order to examine as much surface as possible in the given time frame with maximum measurement accuracy. SimCA* achieves this task by calculating the value of the actuation signal, which represents the portion of time each sensor $\{S1, \ldots, S5\}$ is used during every adaptation period. As an indication of the complexity of the data used in the evaluation: the total number of sensor configurations that can be selected in the UUV case is $5^5 \cdot 10^6$.

The controller pole $p_i$ is set to values between 0.6 and 0.9 which allows to reject errors/disturbances of high magnitude; the choice of concrete pole values is discussed in Section 7.6. $\delta$ is kept at a default value: $\delta = (\max_i - \min_i) \cdot 0.05$.

The application collects the UUV data to build performance graphs, which are used to evaluate SimCA* in the following sections. The $x$-axis of the graphs are time instants $k$. Thus, the $y$-axis shows the average values of the measured feature per 100 surface measurements of the UUV system.
Adaptation with STO-reqs

Figure 7.6 shows the adaptation results of SimCA* on the UUV system configured according to Table 7.1 and requirements set according to UUV scenario (Section 7.3); the controller pole $p$ is set to 0.6. Adaptation starts with the Identification phase that is clearly visible when $k$ is between 0 and 21. The Control Synthesis phase, immediately followed by the Goal Transformation phase, starts after the relationship between control signals $u_i(k)$ and system outputs $O_i(k)$ is identified ($k = 22$). For comparison, the “Scanning Speed” plot contains an additional line (see “Threshold” in Figure 7.6) depicting requirement $R_1$ as if it was an S-req, i.e., it shows the scanning speed required to monitor exactly 100 km of surface within 10 hours using the available energy pool. As in our case $R_1$ is a T-req, i.e., the UUV must scan $S \geq 100$ km, during the Goal Transformation phase SimCA* finds a combination of sensors that allows to scan more surface using the same energy without losing accuracy and updates the scanning speed goal from 2.7 to 3.2 m/s.

After the goal is updated, the Operation phase starts (from $k$ equals 22 onwards). The two upper plots in Figure 7.6 show that during Operation the system is stable, i.e., the measured energy consumption and scanning speed follow their goals. To demonstrate how SimCA* deals with requirement changes, we adjust the available energy twice: at $k = 100$ from 5.4 to 5.0 MJ and at $k = 170$ from 5.0 to 5.1 MJ. Both adjustments trigger the Goal Transformation phase where the scanning speed is updated according to new conditions. Note that lowering the amount of available energy at $k = 100$ increases the scanned distance (speed changes from 3.2 to 3.55 m/s) but decreases the measurement accuracy.

Figure 7.6 also shows how SimCA* reacts to changes in sensor parameters and sensor failures. At $k = 220$, the energy consumption of sensor $S_1$ increases from 170 to 190 J/s. To compensate for this overconsumption, a portion of time $S_1$ was used is given to a less consuming sensor $S_3$, see the “Sensor usage” plot. However, at $k = 290$, $S_3$ stops working and is replaced by sensor $S_4$, while the measured energy consumption and scanning speed of the UUV remain on the required level.

The experiment ends at $k = 360$, i.e., after 10 hours of time. Over a series of 50 experiments, we measured the following outcomes: the total distance scanned is $121.3 \pm 0.32$ km, the amount of consumed energy is $5.1 \text{ MJ} \pm 135 \text{ J}$, the measurement accuracy is $89.94 \pm 0.04\%$.

Adaptation Guarantees and Trade-offs

In order to experimentally verify the guarantees and different property trade-offs provided by SimCA*, we perform the same experiment with STO-reqs using controllers with pole $p = 0.9$, see Figure 7.7. After 50 runs we got the following results: total distance scanned is $121 \pm 0.28$ km, the amount of consumed energy is $5.1 \text{ MJ} \pm 170 \text{ J}$, the measurement accuracy is $89.94 \pm 0.04\%$.

From the Figures, it can be observed that in both scenarios the systems are stable, have a zero steady-state error and converge to their goals without overshooting. The results confirm that the system requirements are satisfied.
7.6 Experimental Evaluation

As described in Section 7.5, adaptation with SimCA* is influenced by the values of the pole $p$. First of all, a smaller pole leads to a shorter settling time. In particular, the settling time $\bar{K}$ of every controller $C_i$ depends on the pole $p_i$ and a constant $\Delta s_i$ chosen by the system engineer: $\bar{K} = \frac{\ln \Delta s_i}{\ln p_i}$. According to [80, p.85], the commonly used value of $\Delta s$ is 0.02 (2%). Hence:

$$\bar{K}_{0.6} = \frac{\ln |0.02|}{\ln |0.6|} = 7.66 \quad \bar{K}_{0.9} = \frac{\ln |0.02|}{\ln |0.9|} = 37.3$$

These values show the amount of adaptation steps required to obtain a change of amplitude 1 in the measured value of a goal. For example, this guarantee can be observed at $k = 100$ on the “Scanning Speed” plot of both figures where the speed is required to change from 3.14 to 3.58 m/s (change of amplitude 0.44). Then, $\bar{K}_{0.6} = 7.66 * 0.44 = 3.4$ steps and $\bar{K}_{0.9} = 37.3 * 0.44 = 16.4$ steps. These values explain why the measured scanning speed makes almost a vertical jump at $k = 100$ in Figure 7.6, while in Figure 7.7 it takes 17 adaptation steps to converges to a target value.

By comparing the experiment outcomes obtained throughout 50 runs, it can be concluded that a smaller pole leads both to a bigger scanned distance and a smaller error in the energy consumption with the same scanning accuracy. This property of SimCA* can be explained by the fact that a higher settling time makes the system waste more resources in a transient phase.
However, lowering the pole is not always a better option as it leads to weaker disturbance rejection. Due to a small noise amplitude, in the tested scenario both controller successfully rejected disturbances. This may not be the case in real operating conditions, where a UUV is influenced by underwater streams, pressure, etc. Besides, a smaller pole makes the adaptation mechanism react faster not only to goal changes but to disturbances as well. This property can be observed, for example, by comparing the usage curve of sensor $S_2$. In Figure 7.7 it is smoother and has a much lower spike at $k = 220$ than in Figure 7.7. In this case slower reaction may be a benefit as it allows to switch less between different sensor combinations.

**Requirement Change Scenario: Energy Leak**

Figure 7.8 presents the adaptation results of SimCA* during an energy leak at runtime. First, the system is configured and starts working according to UUV scenario (Sections 7.3 and 7.6) in a normal operating mode $M_1$; the pole $p$ is set to 0.75. However, at some point in time during operation ($k = 60$ in this case) the system detects an unexpected drastic loss of energy. As UUV is a very expensive equipment, the priority of the system becomes not running out of energy during operation. As a result, the accuracy requirement $R_3$ is deactivated as it cannot be addressed anymore. From the other hand, the mission could still be completed, so requirement $R_1$ remains in the system and the UUV enters the mode $M_2 =$

![Figure 7.8: UUV adaptation during energy leak, $p = 0.75$.](image)
7.6 Experimental Evaluation

After the energy leak, the system requirements become:

(i) $R_1$: a segment of surface over a distance of bigger or equal to $S$ (100 km) should be examined by the UUV within a given time $t$ (10 hours);

(ii) $R_2^*$: subject to $R_1$, the energy consumption of the vehicle should be minimized.

It is evident from figure 7.8 that after the energy leak ($k = 60$ onwards) SimCA* adapts the system to use the least energy consuming sensor $S_5$, which also allows to keep the vehicle at a high speed, thus addressing $R_1$. However, due to a high utilization, $S_5$ breaks at $k = 130$. At this point, the UUV starts using $S_4$ that is the least energy consuming sensor after $S_5$. $S_4$ has lower scanning speed than $S_5$ (see how scanning speed decreases at $k = 130$), but it is still enough to address $R_1$. At $k = 220$ the requirement $R_1$ changes to $S \geq 114$ km. To cope with the updated goal, the UUV is forced to use sensor $S_2$ leading to an increase in energy consumption. Note that after $k = 220$ the “Goal” and “Threshold” lines coincide on the Scanning Speed plot. It means that the vehicle will be able to scan exactly 114 km of surface and not more, while minimizing the consumption of remaining energy.

As in the previous case, the experiment ended after 10 hours of time. In total, the UUV consumed 3.79 MJ of energy and scanned 113.97 km of surface with 73% accuracy. After a series of experiments, we can conclude that SimCA* copes with the energy leak independent of point in time it happens. We also note that when a sensor failure occurs, the transition to a new goal value can be not as smooth as during the experiment presented in Figure 7.8. The reason for such behavior is that sensor failures occur in between adaptation actions (recall that adaptation period is 100 surface measurements). Thus, until next adaptation action, a random sensor is working instead of the broken one, making the system behave not according to the goals set by SimCA*.

New Requirement Scenario: Fail Rate

In this scenario, a new T-req is activated during system operation, see adaptation results in Figure 7.9. As previously, the system starts working according to the UUV scenario in mode $M_1$; the pole $p$ is set to 0.9. At $k = 75$ the UUV enters a deep water zone where sensors may fail to provide measurement. According to specification, each sensor has a certain fail rate in a deep water, namely: $S_1 = 0.01\%$, $S_2 = 0.06\%$, $S_3 = 0.01\%$, $S_4 = 0.02\%$, $S_5 = 0.04\%$. As high fail rate influences the mission outcomes, an extra requirement $R_4$ is activated and the UUV enters the mode $M_3 = R_1, R_2, R_3, R_4$, with $R_4$: The average failure rate should be $F \leq 0.02\%$.

Figure 7.9 shows that a new Identification phase is started as soon as T-req is activated ($k$ between 75 and 86). The aim of this phase is to create a model for the Fail Rate goal. After that a Fail Rate controller is added to the system and a new inequality is added to Simplex, which leads to a usual Operation phase ($k = 87$ onwards). As in the previous experiments, we change requirement $R_1$ at $k = 100$ and 170; we also shut down $S_4$ at $k = 215$ and increases the energy consumption of $S_1$ at $k = 220$. In response to all of these changes SimCA* works as expected and selects an appropriate combination of sensors to be used, see Figure 7.9. For
Figure 7.9: New requirement activated at runtime, $p = 0.9$. 
example, when $S_4$ stops working, the UUV is forced to use sensor $S_5$ and more of $S_3$.

**Scalability of SimCA**

To demonstrate the scalability of SimCA we significantly increase the number of actuation options (combinations of sensors) by equipping the UUV with two sensor panels. Each panel is provided with 5 on-board sensors that monitor a surface equal to the surface monitored by the single panel in the original case. The panels simultaneously monitor the respective surface, hence, a combination of two sensors (one from each panel) is used at the same time. The sensors have characteristics similar to those in Table 7.1. Due to space constraints, we refer to the project website for a detailed overview of the parameters of sensor combinations (energy consumption, scanning speed, and accuracy). The task of SimCA is to choose among 25 sensor combinations in order to satisfy the following goals: (i) R1: The underwater vehicle must examine $S \geq 210 \text{ km}$ of surface within a period of $t = 10 \text{ hours}$ (i.e., the scanning speed $= S/t \geq 5.83 \text{ m/s}$); (ii) R2: The amount of available energy $E$ is limited to 5.3 MJ (i.e., mission energy consumption $= E/S = 147 \text{ J/s}$).

Figure 7.10 shows results of a scalability scenario with 2 sensor panels working in parallel. The sensor data, as in the previous experiments, is subject to random
SimCA*: a Control-Based Approach to Handle New and Changing Requirements

disturbances of small amplitude. Note that the “Sensor Combination Used” plot shows only the 6 most used combinations for this scenario.

In general, the system shows the same adaptation behavior as in the case of a single sensor panel (converging to a goal value, adaptation to changes. etc.). In particular:

1. At \( k = 75 \) the sensor combination \( S3+S7 \) stops working leading to a switch of sensors being used;
2. The change of goals at \( k = 100 \) and 160 switches the sensor combination of the optimal solution;
3. At \( k = 280 \) we repeat the energy leak scenario leading to the use of combination \( S5+S10 \) that consumes the least amount of energy.

As SimCA* has the scalability properties of simplex, we can conclude that increasing the number of on-board sensors will not change the adaptation outcomes.

**Threats to Validity**

SimCA* can handle one class of adaptation problems (satisfying multiple STO-reqs and adapting to requirement changes), but this class of problems apply to a significant number of software systems. At the same time, the approach should not be used to systems undergoing drastic changes in their behavior at runtime as continuous re-identification is very costly. SimCA* works with STO-reqs that can be transformed into quantifiable goals, which may not be easy for all properties; an example is security. SimCA* cannot handle conflicting requirements that lead to unfeasible solutions (e.g., to satisfy R1, the system is forced to ignore R2). However, when requirement are interrelated (e.g., increase in R1 leads to decrease in R2), the solution will be found if feasible.

We evaluated SimCA* in two domains, focusing on adaption for a typical set of stakeholder requirements (resource usage, performance, reliability, cost). While these systems can be considered as representative instances of a significant family of contemporary software systems, further evaluation is required to validate SimCA* for other types of systems. In the experimental setting we have used only some types of disturbances (e.g., sensor failures and noise) and considered particular scenarios with changing requirements. Understanding the impact of other types of disturbances and other adaptation scenarios on SimCA* requires additional evaluation. We also used simulated systems for evaluation, which is inline with the evaluation conducted by others such as [57, 33, 29]. However, the deployment of SimCA* in a real-world setting is required to confirm the obtained results in practice.

**7.7 Conclusions**

In this paper we presented SimCA* an approach that allows building self-adaptive software systems that satisfy multiple STO-reqs, can handle changes of requirements at runtime, and achieve robustness to environmental disturbances and measurement inaccuracy. SimCA* provides guarantees for the adaptation results. The
effectiveness of SimCA* was formally evaluated and demonstrated on two cases with strict goals.

SimCA* contributes towards the application of formal techniques to adapt the behavior of software systems, which is one key approach for providing guarantees. At the same time, by automatically building a control mechanism that adapts the software, SimCA* does not require a strong mathematical background from a designer, which is a key aspect to pave the way for software engineers to use the approach in practice.

In future research, we plan to extend SimCA* to handle architecture reconfigurations at runtime and to apply the approach in real-world scenarios.
Appendix: Evaluation Scenario 2: TAS

To show the generality of SimCA*, we evaluate the approach with a second case: the TAS exemplar [153]. TAS is a service-oriented application that provides remote health support to patients. The main goal of TAS is to track a patient’s vital parameters in order to adapt the drug or drug doses when needed, and take appropriate actions in case of emergency. To satisfy this goal, TAS combines three types of services in a workflow, shown in Figure 7.11.

For service-based systems such as TAS, the functionality of each service can be implemented by multiple providers that offer services with different quality properties: reliability, performance, and cost. The system design assumes that these properties can be quantified and measured. E.g., reliability is measured as a percentage of service failures, while performance is measured as the service response time. At runtime, it is possible to pick any of the provided services.

We consider that five service providers offer the Medical Service, three providers offer the Alarm Service and only one provider offers the Drug Service. Table 7.2 shows example properties of available services based on data from [33].

The properties of TAS depend on the choice of concrete service providers that process user requests. For example, invoking $S_1$ and $AS_1$ will lead to the failure rate $TAS_{FR} = S_1_{FR} + AS_1_{FR} = 0.36\%$, while invoking $MS_2$ and $D$ will lead to the failure rate $TAS_{FR} = S_2_{FR} + D_{FR} = 0.22\%$.

The system requirements are the following:

Table 7.2: Properties of all services used in TAS.

<table>
<thead>
<tr>
<th>Service</th>
<th>Name</th>
<th>Fail.rate, %</th>
<th>Resp.time, time units</th>
<th>Cost, $\epsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Medical Service</td>
<td>0.06</td>
<td>22</td>
<td>9.8</td>
</tr>
<tr>
<td>S2</td>
<td>Medical Service</td>
<td>0.1</td>
<td>27</td>
<td>8.9</td>
</tr>
<tr>
<td>S3</td>
<td>Medical Service</td>
<td>0.15</td>
<td>31</td>
<td>9.3</td>
</tr>
<tr>
<td>S4</td>
<td>Medical Service</td>
<td>0.25</td>
<td>29</td>
<td>7.3</td>
</tr>
<tr>
<td>S5</td>
<td>Medical Service</td>
<td>0.05</td>
<td>20</td>
<td>11.9</td>
</tr>
<tr>
<td>AS1</td>
<td>Alarm Service</td>
<td>0.3</td>
<td>11</td>
<td>4.1</td>
</tr>
<tr>
<td>AS2</td>
<td>Alarm Service</td>
<td>0.4</td>
<td>9</td>
<td>2.5</td>
</tr>
<tr>
<td>AS3</td>
<td>Alarm Service</td>
<td>0.08</td>
<td>3</td>
<td>6.8</td>
</tr>
<tr>
<td>D</td>
<td>Drug Service</td>
<td>0.12</td>
<td>1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Requirements: min 30 9
R1. The average cost for invoking the TAS service is set to 12¢;
R2. The average response time should be below 35 $tu$;
R3. Subject to R1 and R2, the failure rate of TAS should be minimized.

All requirements must be satisfied during normal system operation mode $M_1 = \{R1, R2, R3\}$. Unlike the UUV case, the TAS is expected to run continuously. The requirements R1-R3 and the properties of the services may change at runtime and the system should adapt accordingly. The adaptation task is to decide, for each request with a patient’s vital parameters, which combination of services to select such that the requirements are satisfied.

The TAS case is realized based on the TAS exemplar [153]. The results of SimCA* applied to a TAS scenario are shown in Figure 7.12. The adaptation works as intended: the T-req is addressed by changing the Response time goal to 31 $tu$ (at $k = 21$), system outputs follow the goal change at $k = 150$, the optimal solution is changed when $S2$ stops responding at $k = 225$. The cost goal is deactivated at $k = 295$ and the system enters mode $M_2 = \{R2, R3\}$, where
TAS uses services $S_5$ and $A_S3$, because they have the lowest response time and failure rate.

The TAS case confirms the results obtained with the UUV study. It supports the generality of the approach by showing that SimCA* is effective in adapting software systems independent of concrete goals or software components that take part in the adaptation.
Chapter 8

Conclusions

In this final Chapter we draw conclusions from the conducted research. In particular, the research contributions are presented in Section 8.1, Section 8.2 outlines directions for future work, and Section 8.3 describes closing reflections.

8.1 Contributions

In this thesis, we addressed the problem of guaranteeing the satisfaction of different stakeholder requirements (STO-reqs) in the presence of disturbances and goal changes. Unlike other control theoretical approaches that solve specific problems within a certain domain (typically focusing on resource allocation or admission control), we introduced a reusable approach for adaptation of application software and middleware services.

Concretely, this thesis contributes to the state-of-the-art with:

- A systematic survey of the CBSA research area that provides a comprehensive and structured view on the use of control theory to design self-adaptive software systems [1]. The survey answers RQ1 by describing trends of research on control-theoretical adaptation of software at the application and middleware level, model paradigms and adaptation solutions used in CBSA, and types of goals and guarantees achieved with the state-of-the-art CBSA approaches.

- SimCA*: a reusable control-based engineering approach that allows to build self-adaptive software systems that satisfy STO-reqs under disturbances [137, 136, 2]. The approach can also deal with adjustment/activation/deactivation of goals at runtime. SimCA* answers RQ2 with a formal model of software systems and an adaptation solution combining controllers with the simplex algorithm to handle STO-reqs, deal with disturbances and goal changes. The formal and experimental evaluation of SimCA* also answers RQ3. The formal evaluation includes verification of the following guarantees: stability, absence of overshoot, low settling time, robustness to disturbances, zero steady-state error, tracking of infeasible and unbounded solution. During the experimental validation we applied SimCA* to systems with strict requirements (the UUV system and the TAS exemplar), confirm that the guarantees hold in those systems and analyze the achieved software qualities, such as performance, reliability, cost, etc.
8.2 Future Work

Future efforts within the line of research presented in this dissertation could include two major steps. As we used only informal case studies with simulated environments in this Thesis, the first step would be performing a full-scale case study with the approach in a real-world setting. The second step would enhance SimCA* with an ability to deal with architecture reconfigurations of software at runtime, as currently SimCA* cannot deal with invasive changes of software (e.g., new components entering the system) that may happen in many practical software systems. These two steps are discussed in more details below.

Evaluation in a Real-World Setting

While two informal case studies with simulated environments (UUV system and TAS exemplar) were used to evaluate SimCA* in this Thesis, a full-scale case study in a real-world scenario could be used to strengthen the validity of the research results, and to provide evidence for the applicability and reusability of SimCA*. One of such scenarios that we aim to implement in the future is the deltaIoT system [143], an Internet of Things (IoT) network for monitoring and control of elements in the environment, see Figure 8.1.

The deltaIoT system consists of tiny embedded computers (motes) combined in a network; each mote is equipped with physical sensors monitoring different properties of the elements in the environment (e.g., occupancy or temperature) and a wireless networking unit for communicating results to the neighbor motes or a central control unit (gateway). Some of the motes do not reach the gateway directly, so they use neighbor motes that are closer to the gateway as an intermediate communication step. Therefore, the deltaIoT system is a multi-hop network. Each communication consumes a certain amount of mote’s battery power and has a chance of failing the delivery of data (packet loss). A mote can be tuned to consume more power in order to decrease the packet loss. Each mote also contains a limited queue for storing the incoming packets, so sending too many messages to the same mote will lead to packet drops (queue loss).

As IoT networks are required to work reliably for many years without battery replacement, the main goal in the deltaIoT system is to minimize the total battery power consumption, while ensuring a high packet delivery, i.e., that the packet loss and queue loss are both below certain thresholds. This goal can be achieved by tuning individual power setting on the motes (marked Power on Figure 8.1) and by changing the data communication routes (marked Distribution).

Nowadays, the realization of IoT projects, such as deltaIoT, in different industries is growing with a high pace. Examples include smart homes, smart grids and even smart cities. However, the implementation of IoT projects is challenging because:

- The capabilities of basic network components (motes) are limited as they are typically required to be small and cheap. From the other hand, these components are required to provide reliable communication and work without maintenance for many years. Designing a reliable IoT network that efficiently
8.2 Future Work

Figure 8.1: The deltaIoT system

uses the available energy is a particularly important challenge, as today most of the IoT device energy is spend on communications rather than computation or storage [6].

- It is hard to determine the optimal system configuration, as IoT networks are influenced by a number of uncertainties at runtime. These uncertainties include interferences in the environment that disrupt communications, sudden changes in traffic load, mote malfunctioning or even crashing, among others. Many of the current solutions deal with uncertainties in IoT inefficiently, either by tuning the network manually or by over-provisioning it.

To summarize, IoT networks are expected to meet strict requirements of stakeholders in an uncertain and constantly changing environment, creating the need for self-adaption with guarantees. Hence, IoT is becoming an emerging domain for CBSA research. Combining SimCA* with the deltaIoT system would be one of the future steps of this research.

Architecture Reconfigurations

Currently, we may distinguish between three types of architecture reconfigurations of a software system:

- Structural change: the structure of links between the system components changes. For example, in the deltaIoT system this change is triggered by mobile nodes that change their physical location (and hence links to their neighbor nodes) during system operation.

- Addition of an entirely new component. In this case it is assumed that the system never faced components of this type. For example, a signal amplifier may be added to the current deltaIoT network. It would consume energy to
reduce the noise in the communication channel between the nodes in a certain radius around it.

- Evolution: some components are replaced with the new ones, software interfaces and links between some components change. Such change typically happens during a global software update.

Dealing with architecture reconfigurations is a challenging task. Theoretically, with additional modifications, SimCA* would be able to handle structural change by adding a controller and updating the system of equations solved by Simplex. However, the question of correct system identification and guarantees provided by such modification remains open. To deal with the addition of an entirely new component or an interface change, SimCA* would require a solutions that automatically constructs the system of equations solved by simplex. Implementing such a solutions would be extremely challenging.

8.3 Closing Reflections

In the course of the conducted research, we gained experience in applying control theory to design self-adaptive software systems. Here, we report some closing reflections.

CBSA as a Research Field

The research presented in this Thesis was conducted within the CBSA research field. We have observed that the research interest in this field started growing only during the past couple of years. In CBSA the knowledge is still not well systematized, adaptation solutions for typical problems such as the research problem RP started being explored only recently, and even some obvious questions, such as the connection between control-theoretical guarantees and software quality properties, oftentimes remain without an answer. Hence, there are many open research directions in CBSA that could be explored in future. Also, most of the CBSA solutions suggested in the literature are implemented in simulated environments, so industrial involvement could be very beneficial for this growing research field.

Keep It Simple

Even though nowadays software systems are considered very complex and tend to demonstrate non-linear behavior, we have used simple linear discrete models to describe software behavior in our research. We have found evidence that, even being less accurate, these models are more effective at runtime than complex non-linear models. Our experience confirmed a common view on the use of linear models, stating that the model should capture the system behavior in general, while deviations from this behavior can be addressed by updating the model or the controller at runtime [65, 105]. In addition to that, building non-linear and continuous models is extremely challenging and currently we lack the necessary tools for model identification even in classical control of physical systems. Developing such tools for software systems may require a serious effort.
The Guarantees

In principle CBSA approaches provide guarantees “by design” using the system model. If the model does not represent the system behavior sufficiently then the controller cannot provide guarantees or even achieve the required goals. That is why in our work the model was learned and updated at runtime using the data from a real operating environment. In practice, such model showed to be representative enough to obtain the guarantees in the running software system.

Another aspect that we experienced during our work was to identify how some of the control-theoretical guarantees, such as stability and overshooting, influenced different types of software qualities, such as performance and reliability. This happened because system engineers generally try to avoid instability and non-zero overshooting when designing a system. So it becomes impossible to compare a system with and without overshooting as the former does not exist. The solution we used in this case was the knowledge of how overshooting influences any system output in general: it makes the measured value surpass the goal value in a transient state. Reflecting this back on the software qualities, we concluded that the system with overshooting on, e.g., the energy consumption goal would over- or under-consume energy in a transient state. However, it was still challenging to tell how much energy exactly would be over- or under-consumed.
Bibliography


