Stepan Shevtsov

A Control-theoretic Approach to Realize Self-adaptive Software Systems with Guarantees

Doctoral Thesis
Computer Science and Media Technology
Advisor: Danny Weyns
Co-advisor 1: Martina Maggio
Co-advisor 2: Jesper Andersson
2019

Linnaeus University

Katholieke Universiteit Leuven
A thesis for the dual degree of Doctor of Computer Science from Linnaeus University and Doctor of Engineering Science (PhD): Computer Science from KU Leuven

A Control-theoretic Approach to Realize Self-adaptive Software Systems with Guarantees

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

Linnaeus University
Faculty of Technology
http://www.lnu.se/
Växjö, Sweden, 2019

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 also like to thank my second co-supervisor, associate professor Jesper Andersson for his support.

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 like to thank Katholieke Universiteit Leuven and especially the DistriNet group for providing opportunities for research cooperation and supporting my work with the Internet of Things systems.

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 using Simplex, which I continuously did in this Thesis.

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

Växjö, Sweden
January 6th, 2019
Abstract

Engineering modern software systems is a challenging task as these systems are subject to different types of uncertainties. Examples of such uncertainties are disturbances in the environment that are difficult to predict and goals that may change during operation. The idea of self-adaptation is to handle these uncertainties at runtime, when the knowledge becomes available to resolve them. As more software systems with strict requirements are designed to be self-adaptive, the need for adaptation guarantees is becoming a high-priority concern. Providing such guarantees with traditional architecture-based approaches has shown to be challenging, calling for new approaches to engineer self-adaptive systems. To tackle this challenge, this thesis studies control-based software adaptation (CBSA). CBSA applies principles from control theory to design self-adaptive software systems. More specifically, we address the following research problem using CBSA: how to realize self-adaptive software systems that satisfy multiple stakeholder requirements with guarantees in the presence of uncertainties.

The thesis addresses the research problem in two subsequent stages. The first stage focuses on satisfying multiple stakeholder requirements of different types, and providing adaptation guarantees. This stage starts with a systematic literature review of CBSA, which provides a comprehensive overview of the field, including existing CBSA approaches, applied models and controllers, and analyzed guarantees. From the review, we identify a number of gaps in the existing research and concrete challenges in addressing the research problem. Then, we devise SimCA, a control-theoretic approach to realize self-adaptive software systems that satisfy multiple requirements with guarantees. SimCA combines mathematical models of software system, a control-based adaptation mechanism, and formal analysis of the required guarantees. SimCA is also reusable, meaning that it can be applied to a family of cooperative software systems with strict requirements.

The second research stage focuses on handling different types of uncertainty. We first discuss the types of uncertainty and study whether existing CBSA approaches try to deal with these types. We then introduce an enhanced approach called SimCA* that includes components to deal with uncertainty in software parameters, addition or removal of requirements at runtime and software component interactions. In order to obtain evidence about the applicability and reusability of SimCA and SimCA*, we apply informal exploratory case studies with three software systems with strict requirements from different domains.
Contents
Research Overview 1
1 Introduction 2
  1.1 Problem Definition 3
  1.2 Scientific Approach 4
  1.3 Scope of Research 6
  1.4 Contributions 8
  1.5 Thesis Overview 9
2 General Research Background 11
  2.1 Self-adaptive Software Systems 11
  2.2 Control theory and software systems 13
  2.3 Guarantees 14
Stage I. Satisfying Requirements 17
3 Overview of Research Stage I 18
  3.1 Background 18
  3.2 Research Problem 21
  3.3 Research Approach 22
4 A Systematic Literature Review on Control-Based Software Adaptation 27
  4.1 Introduction 28
  4.2 Focus of the Literature Study 30
  4.3 Self-adaptive Systems Background 33
  4.4 Control Theory Background 35
  4.5 Related Efforts 38
  4.6 Research Method 40
  4.7 Result Analysis 50
  4.8 Discussion 71
  4.9 Threats to validity 75
  4.10 Conclusion 77
5 SimCA: a Control-theoretic Approach to Adapt Software Systems 79
  5.1 Introduction 80
  5.2 Motivating scenario: UUV System 82
  5.3 Simplex Control Adaptation 84
  5.4 Evaluation of Guarantees 90
  5.5 Experimental Evaluation 93
  5.6 Related Work 103
  5.7 Conclusions 104
6 SimCA: Handling New and Changing Requirements 105
  6.1 Introduction 106
  6.2 State of the art overview 108
  6.3 Problem Definition 108
  6.4 Simplex Control Adaptation* - SimCA 110
  6.5 Formal Evaluation of Guarantees 116
  6.6 Experimental Evaluation 118
  6.7 Conclusions 127
Stage II. Handling Uncertainties 131
7 Overview of Research Stage II 132
  7.1 Background: Uncertainty 132
  7.2 Research Problem 132
  7.3 Research Method 133
8 Overview of Control-Theoretical Approaches to Realize Self-adaptation 137
  8.1 Introduction 138
  8.2 Background 139
  8.3 Automated Control-Theoretical Software Adaptation 141
  8.4 Challenges 152
  8.5 Conclusions 153
9 SimCA*: Handling Uncertainty in Self-Adaptive Software Systems 155
  9.1 Introduction 156
  9.2 Focus of Study 159
  9.3 State of the art overview 163
  9.4 Problem Definition 165
  9.5 Overview of Simplex Control Adaptation* – SimCA* 168
  9.6 Basic SimCA and Dealing with Disturbances 170
  9.7 Handling Uncertainty in System Parameters 174
  9.8 Handling Uncertainty in Component Interactions 175
  9.9 Dealing with Requirements Uncertainty in SimCA* 177
  9.10 Formal Evaluation of Guarantees 179
  9.11 Experimental Evaluation 181
  9.12 Conclusions 194
Research Summary and Road Ahead 197
10 Conclusions 198
  10.1 Contributions 198
  10.2 Future Work 199
Bibliography 201
# List of Figures

1.1 Stages of conducted research; grey areas highlight the main focus of each stage. ................................................. 4

2.1 Architecture-based self-adaptive software system (based on [75]). .......................................................... 12

2.2 Control-based self-adaptive software system ................................................................. 13

2.3 The standard control-theoretical guarantees ................................................................. 15

3.1 The Push-Button Methodology ..................................................................................... 19

3.2 Solving tasks with simplex: a geometric representation .................................................. 21

3.3 The detailed scheme of research stage I ........................................................................ 23

3.4 The scheme of a systematic literature review ................................................................ 23

3.5 The analytical method .................................................................................................. 25

3.6 The exploratory case study method ................................................................................ 25

4.1 The three layers representing modern computing systems. The focus of the review is displayed in grey......................................................... 31

4.2 Control-theoretical techniques. In grey the techniques that are considered in this literature review. ........................................................................ 32

4.3 Model of a self-adaptive software system. ....................................................................... 34

4.4 Block diagram of a feedback control scheme .................................................................. 36

4.5 Block diagram of a feedforward control scheme ............................................................. 37

4.6 Control-Theoretical Guarantees. ..................................................................................... 50

4.7 F2: Number of analyzed primary studies by year. .......................................................... 51

4.8 F6: Presentation quality scores. ....................................................................................... 52

4.9 Results for data items required to answer RQ1. .............................................................. 54

4.10 Results for different properties of system models. ......................................................... 56

4.11 F14.1: Sensors. ............................................................................................................... 59

4.12 F14.2: Actuators. ........................................................................................................... 62

4.13 Relation between F14.1 Sensors and F13 System model. ............................................... 64

4.14 Relation between F14.2 Actuators and F13 System model. ........................................... 65

4.15 Results for different properties of system models. ......................................................... 66

4.16 Relation between F16 Controller type and F13 System model. ....................................... 67

4.17 F19 Software qualities. .................................................................................................. 67

4.18 F16: Controller type vs F19: Software qualities. ............................................................. 71

5.1 Identification phase of SimCA. ....................................................................................... 85

5.2 Control Synthesis phase of SimCA. ................................................................................ 87

5.3 Operation phase of SimCA. ............................................................................................ 89

5.4 Properties guaranteed by the controllers ........................................................................ 90

5.5 UUV adaptation with runtime changes. ......................................................................... 94

5.6 The influence of controller pole on SimCA. .................................................................... 96

5.7 SimCA vs Simplex: constant disturbance. ...................................................................... 97

5.8 Detection of Infeasible Solution in SimCA. .................................................................... 97

5.9 SimCA with 25 sensor combinations. ............................................................................. 98

5.10 TAS workflow. .............................................................................................................. 99

5.11 SimCA on TAS scenario. ............................................................................................... 101
Research Overview

The first part of this dissertation provides a generic overview of conducted research. It consists of two chapters:

• Chapter 1 gives an in-depth introduction to the research topic of this dissertation. The chapter defines the problem being tackled and the research scope. It also describes the scientific approaches being used and the research contributions. The chapter concludes with a brief overview of different parts of this dissertation.

• The research background is presented in Chapter 2. 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. Finally, we give an overview of control-theoretical guarantees and their formal analysis.
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 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 e-health systems that provide medical support to patients, unmanned underwater vehicles used for oceanic surveillance, and Internet of Things applications for smart environments.

These systems have some characteristics in common. They have multiple (possibly conflicting and changing) requirements. These requirements are often strict, e.g., an attempt to save energy should not lead to a loss of an expensive underwater vehicle during a mission or a shutdown of patient health tracker in an e-health system. These systems are also expected to deal with changing availability of resources, cope with failures, and autonomously adapt to changes in the dynamic execution environment.

Engineering systems with strict requirements is a complex task due to different kinds of uncertainties that are difficult to predict before system operation [160]. 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. Engineering such systems is also challenging due to differences in their structure. For example, an underwater vehicle has an autonomous centralized software system that is deployed on board, while the IoT network is a heterogeneous system, where many decisions are taken locally on each of the network nodes.

More than a decade ago, a seminal paper of Kephart and Chess [98] 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 a set of high-level goals. This statement have led to the development of a whole new class of systems that are known today as self-adaptive systems. The idea is to let the system autonomously handle different kinds of uncertainties during system operation, when the knowledge to deal with those uncertainties becomes available [43, 51]. 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.
Today, strict requirements of many software systems, such as the examples
given above, have lifted up the requirements for self-adaption. These systems
need guarantees for the adaptation results, such as system stability, robustness to
disturbances and others [35, 42, 163]. One of the principled techniques to real-
ize self-adaptive systems that provide guarantees is \textit{control theory}. Control the-
ory 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 adap-
tation of computing systems [139, 117], a notable one being the book of Heller-
stein et al. [82]. However, most of these efforts concentrated on resource alloca-
tion or admission control solutions. Those solutions are fundamentally different
from adaptation of software systems, because of software complexity and domain
dependency of software context, goals and requirements (which can change at run-
time), among other reasons [50, 24, 67, 160].

Control-based software adaptation (CBSA) only recently emerged as a research
field [50]. CBSA research is not yet systematized and it is lacking solutions for
adaptation problems posed by the systems with strict requirements. Even though
some of these solutions exist [65, 142, 102], they are designed for a specific ap-
lication 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 control-theoretical approach to realize 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 research problem \textit{RP} can be identified:

\begin{center}
\textit{RP: How to realize self-adaptive software systems that satisfy multiple
stakeholder requirements with guarantees in the presence of uncertainties.}
\end{center}

This research problem is complex and poses a number of challenges. Therefore,
it was decided to tackle it in two subsequent research stages, see Figure 1.1.

In research stage I, we focus on satisfying multiple stakeholder requirements
of different types, and providing adaptation guarantees. At this research stage
we look only at some types of uncertainty, namely at disturbances and require-
ment changes. Here, with \textit{disturbances} we mean changes in the environment and
inaccurate measurements. With \textit{requirement changes} we mean change of require-
ment values at runtime. The solution produced at research stage I should also be
\textit{reusable}, i.e., it should not be tied to a specific system or application domain\textsuperscript{1}. As
such, the research problem \textit{RPI} for research stage I is:

\textsuperscript{1}Note that our solution does not target all types of software systems, but rather a family of systems
with strict requirements working under certain assumptions, see Section 1.3 for details.
Introduction

Multiple, different types

Stage I

Requirements

Uncertainties

Disturbances, requirement changes

Guarantees

Yes

Stage II

Multiple, different types

Disturbances, add/remove requirement, uncertainty in software

Yes

Figure 1.1: Stages of conducted research; grey areas highlight the main focus of each stage.

RPI: “To devise a reusable CBSA solution that guarantees the satisfaction of different types of requirements in the presence of disturbances and requirement changes”

The second stage of research is aimed at handling different types of uncertainty, such as addition or removal of system requirements at runtime, disturbances, changes of system parameters and uncertainty in interactions between software components. Note that this dissertation considers only anticipated uncertainty in requirements, meaning that the system goals that represent non-functional (or quality) requirements (response time, failure rate) can be activated, deactivated, or their values may be adjusted at runtime based on conditions that are defined before deployment, but that can only be resolved during operation.

At the same time, the solution produced at this research stage should preserve the outcomes of research stage I, such as reusability, dealing with multiple requirements and providing the guarantees. Summarizing, the research problem \textit{RPII} looks as follows:

RPII: “To devise a reusable CBSA solution that satisfies multiple stakeholder requirements in the presence of different types of uncertainty, and provides guarantees on the adaptation results”

Each of the two research stages has own research questions, research methods and outcomes. An overview of research stage I is provided in Chapter 3, while its outcomes are available in Chapters 4-6. Research stage II is outlined in Chapter 7, its outcomes are available in Chapters 8-9.

1.2 Scientific Approach

We used different research methods in each of the two research stages. This section gives a brief generic overview of each of these methods and achieved results.

To address the research problem \textit{RPI} 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 detail in Chapter 3 together with a comprehensive scheme of research stage I.
1.2 Scientific Approach

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 different types of requirements (RPI), we performed a systematic literature review (SLR) following the guidelines from [100]. 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, 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 [20]. As a first step of the analytical method, we designed a CBSA approach that deals with RPI, 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.

In order to analyze the developed CBSA approach and the obtained guarantees, we applied informal exploratory case studies. 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 [147]. In our work we use the prefix “informal” as we did not use a real-life study context as suggested in the literature [184, 147]. Instead, we analyzed the approach by performing case studies with two simulated software systems: an unmanned underwater vehicle (UUV) system (based on [151]) performing surveillance mission and a service-based medical assistance system (TAS exemplar [172]). During the case studies, we analyzed software qualities that are satisfied by the developed CBSA approach and experimentally verified the obtained guarantees.

To address the research problem RPII, we used two research methods: descriptive literature review and informal exploratory case studies. These methods are explained in detail in Chapter 7 together with a comprehensive scheme of research stage II.

A descriptive literature review is a type of informal review that aims to summarize existing research publications on a certain topic and find commonalities or gaps in these publications [137]. Our descriptive literature review focused on automated control-theoretical approaches to realize software adaptation that were developed after research stage I, i.e., after we conducted the systematic literature review of the CBSA research field. The review showed that existing automated CBSA solutions tackle only a small part of uncertainties, typically focusing on disturbance rejection and dealing with changes of requirement values.

Using the literature review results, we have built upon our approach that deals with RPI and introduced a new CBSA approach that handles different types of uncertainty, thus addressing the research problem RPII. In order to validate this approach and analyze obtained adaptation guarantees, we again applied informal exploratory case studies. The approach was first verified on the UUV system. Then, in order to observe how the approach deals with uncertainty in interactions between software components, it was tested on a simulated Internet of Things net-
work (DeltaIoT system [164]). Similarly to stage I, we analyzed software qualities and guarantees obtained by the developed CBSA approach.

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.).
- Has multiple possibly conflicting requirements 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.
- Does not have to deal with human-in-the-loop or uncertainty related to ownership of software elements.

While these assumptions put restrictions on the target application domains, they hold for a large family of modern software systems. An example of such system that we consider in our research 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.
1.3 Scope of Research

- 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 with maximum measurement accuracy using a specific pool of energy. These requirements 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.

Quality Properties and Control-Theoretical Guarantees

Typically, a stakeholder requires software to meet certain quality properties, such as high performance or reliability. In this work, we focus on adaption for a typical set of software quality properties [170]:

- 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.

In order to achieve these quality properties, a software must satisfy certain system-specific requirements. For example, to achieve a certain level of performance quality property, the UUV system is required to scan a certain distance, while the TAS exemplar must meet a specific service response time requirement. For other relations between quality properties and system requirements used in our work see Table 1.1.

As for the guarantees, we consider a standard set of guarantees analyzed in control theoretical literature [82, 69, 112, 16]. This set includes stability, over-shooting, steady-state error, settling time, robustness to disturbances and inaccurate measurements (see Section 2.3 for detailed description of these guarantees).


1 Introduction

Table 1.1: Relation between quality properties and system requirements

<table>
<thead>
<tr>
<th></th>
<th>Performance</th>
<th>Efficiency</th>
<th>Reliability</th>
<th>Accuracy</th>
<th>Business value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UUV system</td>
<td>Scanned</td>
<td>Energy</td>
<td></td>
<td>Measur.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>distance</td>
<td>consumption</td>
<td></td>
<td>accuracy</td>
<td></td>
</tr>
<tr>
<td>TAS exemplar</td>
<td>Response</td>
<td>Failure</td>
<td></td>
<td>Service</td>
<td></td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>rate</td>
<td></td>
<td>cost</td>
<td></td>
</tr>
<tr>
<td>DeltaIoT</td>
<td>Energy</td>
<td>Packet &amp;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>consumption</td>
<td>queue loss</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We also look at additional guarantees provided by our approach, such as: solution optimality, scalability, and tracking of infeasible or unbounded solution. All guarantees are provided “by design” meaning that they hold at both design time and during software operation (runtime).

1.4 Contributions

In accordance to the research problem $RP$, this Thesis introduces three major contributions:

- A systematic literature review of the CBSA research area that provides a comprehensive and structured view on the use of control theory to design self-adaptive software systems [2]. The review describes 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. This literature review serves as a foundation for all the following contributions.

- SimCA: a reusable control-based engineering approach that allows to build self-adaptive software systems that satisfy different types of requirements [154, 153, 1]. SimCA directly addresses $RPI$ as it can also deal with disturbances and requirement changes at runtime. SimCA includes a formal model of software systems and an adaptation solution combining controllers with the simplex algorithm to handle multiple requirements. The formal evaluation of SimCA deals with the part of $RPI$ regarding the guarantees. 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.

- SimCA*: a reusable control-based engineering approach that builds upon SimCA and handles different types of uncertainty [156]. SimCA* addresses $RPII$ by introducing new components that deal with uncertainty in software parameters, addition or removal of requirements at runtime and software component interactions. During the experimental validation, we apply
SimCA* to the UUV system and the DeltaIoT network to confirm the approach functionality and obtained guarantees.

1.5 Thesis Overview

This dissertation is structured in 10 chapters grouped into following major parts:

- Research Overview. An introductory part that consists of two chapters. Describes the research topic of this Thesis, the problems being tackled, the research methods used and the research contributions. Provides a generic research background and introduces key terminology.

- Stage I. Satisfying Requirements. A Thesis part dedicated to solving the research problem $RPI$, with a focus on satisfying multiple stakeholder requirements of different types, and providing adaptation guarantees. This part consists of four chapters: an overview of research stage I (including research questions, background and methods), a systematic literature review of CBSA, and two chapters presenting our SimCA approach.

- Stage II. Handling Uncertainties. A Thesis part dedicated to solving the research problem $RPII$, with a focus on handling different types of uncertainty. This part consists of three chapters: an overview of research stage II, a descriptive literature review that studies automated control-theoretical approaches to realize self-adaptive software, and a chapter presenting the SimCA* approach.

- Research Summary and Road Ahead. A concluding part of this dissertation. Consists of one chapter that discusses the research contributions and directions for future work.

A more detailed description of chapter contents is given in the beginning of each of the Thesis parts.
Chapter 2
General Research Background

The goal of this Chapter is to provide a research background. We start with introducing the principles of self-adaptation in Section 2.1 and control theory in Section 2.2. Then, we discuss formal guarantees in Section 2.3.

2.1 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 [43, 51].

A classic approach to engineer self-adaptive software is architecture-based adaptation [75, 135]. 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 2.1 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 [18] or probabilistic temporal logics [33].
General Research Background

Adaptation Goals

Adaptation System

Sense

Software System

Environment
Non-controllable software, hardware, network, physical context

Figure 2.1: Architecture-based self-adaptive software system (based on [75]).

- 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) [98]. 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, model checking) at runtime is difficult to achieve [171]. These insights have led to the exploration of new approaches, such as control theory, for engineering self-adaptive software.
2.2 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” [55] it started being explored as a solution to adapt computing systems. Seven years later, the book by Hellerstein et al. [82] 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 [26, 50].

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 2.2, 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 2.2: Control-based self-adaptive software system](image)

Using the same SBS example from Section 2.1, 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 $R_{\text{goal}}$ 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 2.2 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 [82, 16]. 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 [82]. 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 [179]. PID calculates the control signal to eliminate the control error [15]. 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 2.2.

In this Thesis, we use multiple adaptive PI-controllers $C_{pigm}$ (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 $C_{pigm}$ is described in Section 3.1, Simplex is explained in Section 3.1. 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 2.3.

### 2.3 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 [82], control theory can guarantee four main system properties, see Figure 2.3:

- **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 2.1. 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
violate 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 [82]. 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.
Stage I. Satisfying Requirements

This part of the dissertation describes stage I of the conducted research in details and how we solved the research problem RPI. It consists of the following chapters:

- Chapter 3 provides an overview of research stage I. First, it gives a background on system requirements and describes the two key components of SimCA: the Push-Button Methodology and the Simplex method. Then, the chapter introduces research questions of stage I. Finally, the research methods being used in stage I are described in details.

- 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 two types of system requirements (to keep a value at a required level and to minimize or maximize a value) 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 two types of requirements 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 different guarantees.

- Chapter 6 describes a new version of SimCA that addresses new type of requirements where a value is kept above/below a threshold and can deal with requirement values that change at runtime. Similar to Chapter 5, this chapter includes a formal model and a control solution used for software adaptation, as well as the analysis of qualities and guarantees achieved by systems equipped with SimCA.
Chapter 3

Overview of Research Stage I

This Chapter gives an overview of research stage I. Section 3.1 provides the necessary research background, Section 3.2 outlines the research problem and the research questions to be answered at this research stage, while Section 3.3 describes the research methods being used.

3.1 Background

This Section provides research background for Stage I of the conducted research. We describe STO-requirements, the Push-Button Methodology and the Simplex method in the following subsections.

**STO-requirements**

The initial discussion about software quality properties and related system-specific requirements considered in this dissertation is available in Section 1.3. In this section, we discuss the requirement types.

In order to make the definition of the research problem \( RPI \) more precise, we determined the types of system requirements by looking at existing self-adaptive artifacts\(^1\) and systems with strict requirements from the literature, e.g. [67, 102]. We identified three commonly used types of system 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, [102] defines a particular response time of a web server that the adaptation solution should guarantee (S-req). The automated traffic routing system [182] 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 [76], 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.

\(^1\)http://self-adaptive.org
**Push-Button Methodology**

In this section we summarize the Push-Button Methodology (PBM) [67] which serves as a basic component of 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 [67] 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 3.1:

![Figure 3.1: The Push-Button Methodology](image)

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

\[
y(k) = \alpha \times \eta(k - 1)
\]

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 \( M_{p bm} \) and adapts the software is automatically created:

\[
\eta(k + 1) = \eta(k) + \frac{1 - p}{\alpha} \times e(k + 1)
\]

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 2.3.

To cope with small disturbances and system dynamics, PBM updates the value
of $\alpha$ during system operation using a Kalman filter [168], 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 Sec-
tion 2.1. 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 num-
bers of tasks affect the system response time. Based on this model, PMB cre-
ates 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 num-
ber 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 au-
automatically during model building, while $p$ is chosen by the system engineer).

Simplex

In this section we explain the simplex method which is a second basic component
of 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:

$$\min \{ c^T x \mid Ax \leq b; x \geq 0 \}$$ (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 [47].

To simplify understanding of the problem, we rewrite (3.1) as a system of equa-
tions using a modified SBS scenario from Section 2.1. 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 $F_2$) characteris-
tics 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 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:

\[^2\text{Matrix transpose is an operator that turns all rows of a given matrix } A \text{ into columns and all columns of } A \text{ into rows. The resulting matrix is denoted } A^T.\]
Minimize Failure rate:

\[
\min [F_1 \times x_1 + F_2 \times x_2]
\]

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. 3.2). 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 [47, 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 3.2: Solving tasks with simplex: a geometric representation

3.2 Research Problem

Based on research problem RPI “To devise a reusable CBSA solution that guarantees the satisfaction of different types of requirements in the presence of disturbances and requirement changes”, we identified three research questions of research stage I. As there is no systematic study consolidating knowledge in CBSA,
with research question \textit{QI-1} we want to get trends of research in CBSA and study existing CBSA approaches that may potentially address the research problem \textit{RPI}:

\textbf{QI-1:} \textit{“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 \textit{model} of software and use a \textit{control solution} that adapts software based on this model in order to satisfy the system requirements. Therefore, with research question \textit{QI-2} we want to identify the two key components of a CBSA approach that can address the research problem \textit{RPI}:

\textbf{QI-2:} \textit{“What are the appropriate models and control solutions that can be used to address STO-reqs, and deal with disturbances and requirement changes?”}

Systems with strict requirements prioritize different software qualities. While for a 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 \textit{QI-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 \textit{RPI}:

\textbf{QI-3:} \textit{“What software qualities can be satisfied and what guarantees are provided with a CBSA approach that deals with STO-reqs, disturbances and requirement changes?”}

3.3 Research Approach

In this Section we provide a detailed description of the methods used to conduct the research in stage I, see Figure 3.3. The choice of these methods was based on the research problem \textit{RPI}, the research questions and the specific goals we set during research stage I. In particular, we discuss systematic literature review, analytical method, and informal exploratory case studies in the following subsections. We motivate the selection and give an overview of each research method.

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 \textit{QI-1}, we performed a systematic literature review (SLR) following the guidelines from [100]. A systematic literature review is an established method to collect and analyze data
3.3 Research Approach

**RPI**: To devise a reusable CBSA solution that guarantees the satisfaction of different types of requirements in the presence of disturbances and requirement changes

What CBSA approaches are used to deal with STO-reqs in the existing literature?

What are the appropriate models and control solutions that can be used to address STO-reqs, and deal with disturbances and req. changes?

What software qualities can be satisfied and what guarantees are provided with a CBSA approach that deals with STO-reqs, disturbances and req. changes?

---

**Overview of CBSA**

**Push-Button Methodology**

**SimCA**

**Formal analysis of guarantees**

**Experimental verification**

**Systematic Literature Review**

**Analytical Method**

**Informal Exploratory Case Study**

---

**Legend**

- Provides input
- Research question
- Research method
- Key result

---

**Figure 3.3**: The detailed scheme of research stage I

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 [23]. 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 [100].

An SLR comprises three main phases: planning, executing, and reporting, see Figure 3.4. 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 3.4**: The scheme of a systematic literature review
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 \textit{RPI} or the STO-reqs of software (\textit{QI-1}), 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 [67], 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 \textit{RP} presented in this Thesis, see Figure 3.3.

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

Analytical Method

We used the analytical method to answer research question \textit{QI-2} and a part of research question \textit{QI-3} regarding the guarantees, see Figure 3.3. The analytical method is a research method used to develop, analyze and validate a formal theory or set of axioms [20]; it is typically applied in the formal computer science research [180]. The analytical method consists of four steps, see Figure 3.5:

- 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 [174, 122].

- 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.
3.3 Research Approach

In this Thesis, in the first step of the analytical method we leveraged upon the Push-Button Methodology [67], 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 requirement changes (QI-2). Then, we analyzed the behavior of software systems equipped with SimCA through the application of mathematical techniques and formally verified a number of guarantees (QI-3). 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 6.

Informal Exploratory Case Studies

We used informal exploratory case studies to answer a part of research question QI-3 regarding the software qualities and to experimentally verify the previously analyzed guarantees, see Figure 3.3. 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 [147]. 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 3.6. 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.

According to [184], 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 [184, 147]. 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 thousandths of measurements for free. A full-scale case study in a real-life context is a part of our future work, see Section 10.2.
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 \( (QI-3) \). 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 and 6.
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 QI-1 “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 [2]. The review supporting material can be found in [152]. Personal contribution: Stepan Shevtsov performed 75% of study design and implementation, 60% of data collection, 60% of data analysis, and 60% of writing the manuscript.
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 [136, 98]. 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 [89]. Moreover, software should execute with incomplete knowledge of the execution environment and conditions and face changing requirements during operation [160]. 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 [43, 51].
Different alternative approaches have been proposed for the design of self-adaptive software, a prominent one being architecture-based adaptation [135, 74, 106, 175]. In the architecture-based approach, the software generates and updates an explicit architectural model of itself and uses it to reasons 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 [32, 171].

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 [148]. 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 [26, 82, 50, 189].

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, 53, 190]. With respect to the adaptation of resource allocation, applying control theory to adapt the software behavior is a more complex problem [67, 8, 24], due to the difficulty of accurately modeling software, to the types of requirements and their tradeoffs [9] and to the need of instrumenting software to obtain sensor measurements and actuators [28, 85].

Research efforts applying control-theoretical adaptation to software exist [60, 22, 67, 154]. 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 [100]. 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 \cite{33} 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 4.8. 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 \cite{82, 139} with different objectives. This systematic literature review focuses on software adaptation\(^1\). 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: webservises, 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 \cite{139}, 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 \cite{4, 27}.

\(^1\)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]

<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 [9];
- Context, goals and requirements are domain-dependent and can change during runtime [148, 160];
- There is a necessity to use complex and potentially multiple system models simultaneously [67, 28];
- The choice of proper sensors and actuators for adapting software can be challenging [50];
- The design space for the adaptation of software applications is often multi-dimensional [8, 24];
- 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 [68].

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 [122, 50].
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), $\mathcal{H}_\infty$ 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 [90, 91], 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 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 [167, 80, 133], 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 [78, 119]. 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 [169]. We start with explaining the principles and motivation of self-adaptation. 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 [43, 163, 21, 56, 87, 148].

Principles of Self-adaptation

The term self-adaptation is not precisely defined in the literature. [43] 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” [26] 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.

[74] 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. [8] 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, [169] 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. [98] 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 [33], and fuzzy goals whose satisfaction is represented through fuzzy constraints [18]. 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 [98]). 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. [33] 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 [171]. 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 [112, 16, 110, 134, 115, 111].

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 9.1 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: \( Setpoint + (-Measured\,Output) \). 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. 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% 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. 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).

A particularly successful heuristic for optimal control under constraints is Model Predictive Control (MPC). 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 [129], 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 [112].

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 [87, 170], 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” [55]. Seminal research in this direction is documented in the book by Hellerstein et al. [82], 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
4.5 Related Efforts

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 [81] on mechanisms for performance management of Internet applications and [166] 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 [161].

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. [139]. 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, [139] 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 [102]. 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 [139] 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 [139] do not discuss the guarantees provided by the control-theoretical approaches, while formality is one of the main reasons to apply control theory [134, 16, 82, 26, 50].

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 [100]. 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 [19]:

- **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 assessment 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. [82, 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 [170, 73]. The procedure starts with identifying the document sources that are used in related surveys [170, 139, 50]. 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 4.1: Conferences included in the search, part 1

<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.

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

To validate the search string, we used a “quasi-gold standard” [186]. 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

---

**Table 4.2:** Conferences included in the search, part 2

<table>
<thead>
<tr>
<th>ID</th>
<th>Group</th>
<th>Venue</th>
<th>ARC</th>
<th>H-index</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDC</td>
<td>CT</td>
<td>Conference on Decision and Control</td>
<td>A</td>
<td>45</td>
</tr>
<tr>
<td>ACC</td>
<td>CT</td>
<td>American Control Conference</td>
<td>n/a</td>
<td>40</td>
</tr>
<tr>
<td>ICARCV</td>
<td>CT</td>
<td>International Conference on Control, Automation, Robotics and Vision</td>
<td>A</td>
<td>11</td>
</tr>
<tr>
<td>ECC</td>
<td>CT</td>
<td>European Control Conference</td>
<td>n/a</td>
<td>20</td>
</tr>
<tr>
<td>SASO</td>
<td>AS</td>
<td>Self-Adaptive and Self-Organizing Systems</td>
<td>n/a</td>
<td>9</td>
</tr>
<tr>
<td>Adaptive</td>
<td>AS</td>
<td>Adaptive and Self-adaptive Systems and Applications</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>FeBID</td>
<td>AS</td>
<td>International Workshop on Feedback Computing</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>SEAMS</td>
<td>AS</td>
<td>Software Engineering for Adaptive &amp; Self-Managing Systems</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>SefSAS</td>
<td>AS</td>
<td>Software Engineering for Self-Adaptive Systems</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

---

3 The 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>IJRNC</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. 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

---

4The 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: [102, 123, 57, 103, 52, 66, 65, 12, 67, 84, 68, 153, 140, 141, 138, 3, 94, 22, 5, 109, 108, 104, 60, 63, 61, 37, 128, 30, 29, 93, 92, 38, 185, 142, 165, 127, 39, 9, 154, 10, 96, 95].

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 [55, 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 [58] and adjusted in [173]. 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>
</tr>
</thead>
<tbody>
<tr>
<td>2 Explicit problem description</td>
</tr>
<tr>
<td>1 General problem description</td>
</tr>
<tr>
<td>0 No problem description</td>
</tr>
</tbody>
</table>

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

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

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

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

<table>
<thead>
<tr>
<th>Q6: Limitations of the study</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Explicit list of the study limitations</td>
</tr>
<tr>
<td>1 General words about the study limitations</td>
</tr>
<tr>
<td>0 No description of the study limitations</td>
</tr>
</tbody>
</table>
4.6 Research Method

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 \( t \).
\( o(t) = f(i, x, t) \). 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 [82].

- **F18**: Formal guarantees provided by the use of control theory and described in the study. According to [82], 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 [82], 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 [170], 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 [140, 12, 67, 154].

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>[102]</td>
<td>12.5</td>
</tr>
<tr>
<td>Push Button Methodology</td>
<td>[67]</td>
<td>11.5</td>
</tr>
<tr>
<td>DYNAMICO Reference Model</td>
<td>[165]</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., [170, 73]. 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/controller. Software, in this case, was used as
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 (pre-conditions)</td>
<td>[141, 138, 108, 93, 142, 68, 5, 10]</td>
</tr>
<tr>
<td>Requires additional computation resources or tools (redundancy)</td>
<td>[141, 138, 5, 60]</td>
</tr>
<tr>
<td>Not applicable in some cases/systems</td>
<td>[141, 108, 29, 93, 142, 12, 68, 5, 154]</td>
</tr>
<tr>
<td>External validity: generalising findings requires extra effort</td>
<td>[138, 29, 60, 37, 154]</td>
</tr>
<tr>
<td>Complexity of the proposed approach</td>
<td>[138]</td>
</tr>
<tr>
<td>Not able to handle new requirements</td>
<td>[9]</td>
</tr>
</tbody>
</table>

an application domain, typical software engineering aspects were not well elaborated. The remaining 21 studies [102, 103, 127, 39, 37, 5, 140, 30, 65, 63, 12, 84, 68, 67, 153, 9, 93, 92, 154, 10, 96] 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 [96, 95], 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 [139]. 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 [140] and the RUBiS benchmark\(^1\). Three studies used general web applications that show static content to user [5, 109, 108], while one study used recommender systems [185]. The applications in the video/image processing domain can be divided in different groups: object recognition [104, 60], video streaming [127, 22], video encoding [39, 67, 84], image and signal processing [66, 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. [63, 61, 29] (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 [37, 128], to determine the quality of tests [30, 29], 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 [93], and to analyze the system lifetime based on the amount of development effort [92].

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 [74, 106, 175]. 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 [77]. 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 [14].

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 [67, 65]. 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>[65, 66, 30, 29]</td>
<td>Model used for stochastic systems for which future states depend only on the current state.</td>
</tr>
<tr>
<td></td>
<td>Queueing network</td>
<td>[22, 12]</td>
<td>System is represented as a network of queues, which is evaluated analytically.</td>
</tr>
<tr>
<td></td>
<td>Custom</td>
<td>[61, 93, 92, 104, 60, 96, 95]</td>
<td>Custom analytical models used for object recognition [104, 60], software development process [93, 92], search algorithm [61], ads [96, 95].</td>
</tr>
<tr>
<td>Linear model</td>
<td>Learned at Run-time (LLR)</td>
<td>[102, 123, 57, 103, 52, 67, 153, 84, 68, 154]</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 [102, 67, 153] or using a controller [84, 68]. In some cases [67, 153] $d(k)$ 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>[141]</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>[138]</td>
<td>Models of different types that can inter replace each other during operation depending on the system goals.</td>
</tr>
<tr>
<td></td>
<td>Custom</td>
<td>[5, 108, 37, 128, 140, 39, 3, 94, 9, 185, 10]</td>
<td>Custom grey box models used for web server utilization [5, 108], software testing process [37, 128], resources allocation between software components [140, 39], recommender system [185], component interactions at the application layer [3, 94].</td>
</tr>
<tr>
<td>Black box</td>
<td>Custom</td>
<td>[109, 127]</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>[63, 165, 142]</td>
<td>No specification of the concrete model being used.</td>
</tr>
</tbody>
</table>
elements that can be modeled. Each middleware software or each application has its own technology- and domain-specific software elements. Third, as stated in [67, 68], 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 [138, 109]. 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 [29, 66, 30, 93, 92, 61, 141, 3, 94, 96, 95]. 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 [110, 72].

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 [60, 67, 68, 153, 37, 128, 29] or even switching the model [138], 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 [67, 153, 29]. Other approaches use a separate linear corrector to compensate for model changes [128], allow human operators to make a decision [37], or completely change the control law [60, 138].

**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><strong>Software utility</strong></td>
<td>Probability of correct object recognition</td>
<td>[104, 60]</td>
</tr>
<tr>
<td></td>
<td>Software response to requests</td>
<td>[52, 140, 141, 138, 153, 84, 68, 127]</td>
</tr>
<tr>
<td></td>
<td>User perceived latency of application</td>
<td>[102, 123, 57, 103]</td>
</tr>
<tr>
<td></td>
<td>Video quality and processing speed</td>
<td>[39, 67, 3, 94]</td>
</tr>
<tr>
<td></td>
<td>Profit gained from the application</td>
<td>[92, 142]</td>
</tr>
<tr>
<td><strong>Software inefficiency</strong></td>
<td>Percentage of software failures</td>
<td>[66, 65, 67, 68, 153]</td>
</tr>
<tr>
<td></td>
<td>Detection of software defect</td>
<td>[128, 30, 29]</td>
</tr>
<tr>
<td></td>
<td>Number of errors in software application</td>
<td>[37, 93, 142]</td>
</tr>
<tr>
<td><strong>Resource utilization</strong></td>
<td>Energy consumption</td>
<td>[84, 68]</td>
</tr>
<tr>
<td></td>
<td>Cost of using external services</td>
<td>[68]</td>
</tr>
<tr>
<td></td>
<td>Bandwidth</td>
<td>[5, 109, 108]</td>
</tr>
<tr>
<td></td>
<td>CPU usage</td>
<td>[38, 39]</td>
</tr>
<tr>
<td></td>
<td>Memory usage</td>
<td>[127, 38]</td>
</tr>
<tr>
<td><strong>System utilization</strong></td>
<td>Length of requests queue</td>
<td>[94, 22, 12, 103, 52]</td>
</tr>
<tr>
<td></td>
<td>Amount of new user registrations</td>
<td>[142]</td>
</tr>
<tr>
<td></td>
<td>Data to be processed by the system</td>
<td>[185]</td>
</tr>
<tr>
<td></td>
<td>Request arrival rate</td>
<td>[5, 109, 108, 3, 52]</td>
</tr>
</tbody>
</table>

![Figure 4.11: F14.1: Sensors.](image)

example how the actuation mechanisms implement changes of the application) are discussed in only 7 of the 42 primary studies [5, 108, 102, 68, 63, 165, 39]. [5, 108] describe how bandwidth and request rate of an Apache Web server are measured, [102] employs PHP scripts to control the amount of optional content served to users and calculate user perceived latency, and [63] 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 implementation 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 sensors 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 software 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 elements external to the software application: those that monitor resource utilization and those that monitor system utilization. The sensors that monitor resource utilization 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 inefficiency – 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, programming abstractions, and similar ones) or through a dedicated software interface. 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 respectively 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 [165] and [175].

**Actuators:** As for the actuators, we observed that the studies use a wide variety 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 levels of granularity: parametric, component, and mode adaptation, and architecture reconfiguration.

Parametric adaptation refers to changing the values of variables of the application 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
adaption 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 [105]. 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 [108]. 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 [39, 109]; a fourth study deals with it by minimizing the number of system reconfigurations [127].

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 [66, 65, 67, 84, 68, 153, 60, 61, 127, 38, 154], and only a single study [60] 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 Specific Controller</th>
<th>Primary Studies</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID</td>
<td>Proportional-integral-derivative</td>
<td>[61, 142] A classical controller that is easy to implement and tune. Consists of 3 components (P, I, D) responsible for different controller characteristics [16]. In 13 out of 18 studies, PID controllers are also adaptive as it helps to compensate for inaccuracy and errors in the system model [102, 65, 67]</td>
</tr>
<tr>
<td>PID</td>
<td>Proportional-integral</td>
<td>[123, 102, 57, 103, 67, 153, 138] Proportional [12, 37]. Integral [63, 96]</td>
</tr>
<tr>
<td>PID</td>
<td>Proportional Integral</td>
<td>[12, 37]. [63, 96]</td>
</tr>
<tr>
<td>MPC</td>
<td>Model predictive</td>
<td>[140, 141, 128, 127, 39, 9, 10] 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>MPC</td>
<td>Limited lookahead (LLC)</td>
<td>[94, 22, 3] 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>[185] 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>Feedback+ feedforward</td>
<td>[123] Applied in a single study, where feedforward and feedback controllers are paired and compared to other types of controllers.</td>
<td></td>
</tr>
<tr>
<td>Optimal</td>
<td>Custom optimal</td>
<td>[30, 29] 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>Optimal</td>
<td>Bang-bang</td>
<td>[93, 92] 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>[84, 68, 123] Controller created using a pole placement technique [16, 82].</td>
</tr>
</tbody>
</table>
controller 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” [66, 165, 104, 38]. 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., [57, 103, 3, 94]), 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 [102, 65, 67]. 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 [140, 39]. 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 [187, 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 [57, 103, 3, 94], 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 [153, 154], where multiple controllers at the top level provide inputs to a single controller at the bottom level.

- Two studies apply switching control [138, 60], where different control laws interchange with one another, depending on the actual software adaptation tasks.
Two studies apply cascaded control [84, 68], where the output signal of a high level controller becomes an input for a lower level controller.

Finally, one study applies cooperative control [127], 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,
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, 185, 93]. Seven studies discuss the tuning of a controller to trade different guarantees, typically robustness for settling time [102, 52, 67, 66, 12, 142, 154].

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>[102, 12, 67, 68, 153, 154]</td>
<td>Analysing the pole of the controller.</td>
</tr>
<tr>
<td>Overshoot</td>
<td>[12, 68, 153, 154]</td>
<td>Keeping the pole of the controller in a certain interval.</td>
</tr>
<tr>
<td>Steady-state error</td>
<td>[153, 154]</td>
<td>Analysing the output equation of the system.</td>
</tr>
<tr>
<td>Robustness</td>
<td>[102, 67, 153, 154]</td>
<td>Analysing the feedback loop transfer function.</td>
</tr>
</tbody>
</table>

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 [102, 142]. 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 [68].
### 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, 29, 142, 140, 141, 138]</td>
<td>Ability of the system to achieve its goals. [142] measures stability as the number of system reconfigurations that occur during adaptation.</td>
</tr>
<tr>
<td>Settling time</td>
<td>[52, 140, 141, 138, 95]</td>
<td>The time required to reach the setpoint after a goal change.</td>
</tr>
<tr>
<td>Overshoot</td>
<td>[140, 141, 138, 95]</td>
<td>Spikes in the system output for different adaptation options.</td>
</tr>
<tr>
<td>Steady-state error</td>
<td>[52, 140, 141, 138]</td>
<td>Oscillations in the response time of the software for different adaptation options.</td>
</tr>
<tr>
<td>Robustness</td>
<td>[12, 95]</td>
<td>Deviations in the system output under disturbances.</td>
</tr>
<tr>
<td>Optimality</td>
<td>[127]</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>[39, 109, 108, 84]</td>
<td>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 [52] 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 [67]. Having a more robust software can enhance performance by maintaining low latencies, or increase business value by serving more advertisements on web sites [12].

- **Optimality** is another control property that relates to any type of software quality. Examples in the primary studies are performance [30], security [68], reliability [93], and business value [92]. 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 [84, 127].

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 theoretic 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. [139] and Brun et al. [26], and open challenges for future research in control-theoretical software adaptation.

Before we compare the results of our survey with results reported in [139], it is important to emphasis that the scope of the survey of Patikirikorala et al. [139] is different from our survey: while we concentrate on the adaptation of software, in particular application software and supporting middleware services, [139] 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 [139] 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 [139] (about 65/35). However, a notable distinction concerning model type is that [139] 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, [139] 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 [139]). The most frequently used types of sensors reported in [139] 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 [139].

3) Controller type. As [139] 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 [139]. Optimal control (see LQR in [139]) 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 [139] 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 [139] reported only 15% for this data item. Explaining such a mismatch is not difficult because [139] 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 [139]. As a side note, [139] 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 [139] 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. [26].

Although the article by Brun et al. [26] 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.

[26] 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 [26], 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 [26] 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.

[26] does not consider many aspects that we studied in our systematic literature review (which was not the particular aim of [26]). 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. [66] 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, [12] 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
4.9 Threats to validity

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 realising 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 [139]. 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 [152] 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-theoretic 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 QI-2 “What are the appropriate models and control solutions that can be used to address STO-reqs, and deal with disturbances and requirement changes?” and research question QI-3 “What software qualities can be satisfied and what guarantees are provided with a CBSA approach that deals with STO-reqs, disturbances and requirement 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) [154]. Personal contribution: Stepan Shevtsov performed 100% of the technical implementation (program code, testing, etc.), performed 100% of data collection, 75% of the analysis, and 60% of writing the manuscript.
Keep It SIMPLEX: Satisfying Multiple Goals with Guarantees in Control-Based Self-Adaptive Systems

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 concern. 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 [43, 51]. 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 [163, 35, 42, 174]. Architecture-based approaches for self-adaptation [135, 107, 173], 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 [32, 171], calling for new perspectives on engineering self-adaptive systems.

More than a decade ago, Hellerstein et al. [82] 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., [69, 50]. In this approach, a software system is treated as a plant to be \textsuperscript{1} controlled, and a control feedback loop empowers the software with self-adaptation capabilities, providing formal guarantees, regardless of uncertain operating conditions [25, 64].

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) [67]. 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 at 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 [45] 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 [68], 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-

---

\textsuperscript{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 [68].

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 [151]) 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 $S = 100 \text{ km}$ should be examined by the UUV within a given time $t$ (10 hours in the scenario);
5.2 Motivating scenario: UUV System

Table 5.1: Parameters of sensors of the UUV.

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Energy cons., Scan Speed, Accuracy, on-board sensor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>J/s</td>
</tr>
<tr>
<td>Sensor1</td>
<td>170</td>
</tr>
<tr>
<td>Sensor2</td>
<td>135</td>
</tr>
<tr>
<td>Sensor3</td>
<td>118</td>
</tr>
<tr>
<td>Sensor4</td>
<td>100</td>
</tr>
<tr>
<td>Sensor5</td>
<td>78</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 ±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.
5.3 Simplex Control Adaptation

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:

$$u_i = [\text{min}_i, \text{min}_i + \delta, \text{min}_i + 2\delta, \text{min}_i + 3\delta, \ldots, \text{max}_i]$$

Where $\text{min}_i$ and $\text{max}_i$ 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 $\delta = (\text{max}_i - \text{min}_i) \times 0.05$. 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.

---

$^2$ $u_i$ 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 \cite{121}. 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:

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

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 \cite{67}, 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 \cite{67}. 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:

\[
s_i = \frac{R_i - \Delta R_i}{t_i - \Delta t_i}
\]  

\section*{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 \( \min E = 78 \text{J/s} \), while the maximum is \( \max E = 170 \text{J/s} \). Then, by default, \( \delta = (\max E - \min E) \times 0.05 = 4.6 \). The model identification starts with sending \( u_i(0) = \min E = 78 \text{J/s} \) 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 \( u_i(1) = \min E + \delta = 82.6 \text{J/s} \), 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-
Controller Synthesis Phase

The second phase of SimCA, labeled control 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 [67], the system output equation, representing the measured output $O_i(k)$ in response to a unit step setpoint $s_i$ is defined as follows:

$$O_i(k) = s_i \times (1 - p_i^k) \quad (5.2)$$

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:

$$u_i(k) = u_i(k-1) + \frac{1 - p_i}{\alpha_i} \times e_i(k - 1) \quad (C_i)$$

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 $k$. 

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:

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

$$u_E(k) = u_1(k - 1) + 0.1 \times e_E(k - 1) \quad (5.3)$$

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 $P(k)$,\(^5\) 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:

$$\max\{c^T x \mid Ax \leq b; x \geq 0\} \quad (5.4)$$

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 [47].

\(^5\) $P(k)$ 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 \([47, 46, 144]\).

**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**:

\[
max[Acc_1 \times x_1 + Acc_2 \times x_2 + \cdots + Acc_5 \times x_5]
\]

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; \(Acc_j\) 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 9.2 illustrates these system properties.
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 [82, 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 [47, 46], 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 [101] 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 [45]. 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 [67], 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 [47, 62]: (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 [67].
5.5 Experimental Evaluation

We empirically evaluate SimCA with two cases. First, this Section describes the experimental setting of the UUV case. Then, it 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 and the guarantees provided by SimCA with the case study. After that, the second case with Tele Assistance System is described and evaluated. In addition, we provide a qualitative comparison of SimCA with the approach presented in [68]. Finally, this Section 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 \(\{S1, \ldots, S5\}\) 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 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: \(\delta = (max_i - min_i) * 0.05\).

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.
Figure 5.5: UUV adaptation with runtime changes.
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 consumption 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 \( S_4 \), to satisfy all goals, the system is forced to use \( S_5 \), 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 [82, p.85], the commonly used value of \( \Delta s \) is 0.02 (2%). Hence \( \bar{K} = \frac{\ln 0.02}{\ln 0.9} = 40 \)

\(^1\)http://homepage.lnu.se/staff/daweea/simplex.htm
adaptation steps. This means that changing the scanning speed from 2.7 to 3.1 (step of amplitude 0.4) would take around $\bar{K} = 40 \times 0.4 = 16$ 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 $3.2\, m/s$ 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 $3.6\, m/s$ can be achieved by using $S2$ and $S5$. However, the energy consumption goal of $150\, J/s$ 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 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 \( S = 210 \text{ km} \) of surface within a period of \( t = 10 \text{ hours} \) (i.e., the scanning speed \( = S/t = 5.83 \text{ m/s} \)).

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

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 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 [172]. 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].

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 [172]. 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

### Table 5.2: 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, ¢</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 |
software systems independent of concrete goals or software components that take part in the adaptation.

**Comparison of SimCA with AMOCS**

Recently, [68] 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 [68]:

- SimCA provides guarantees for robustness and steady-state error; AMOCS’ “robustness analysis is system-dependent and unsuitable for an automated control strategy” [68], 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” [68].
- 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” [68]; 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 [68]).
- AMOCS “requires the number of knobs to be greater than or equal to the number of goals” [68]. 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 [68]. 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 δ 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 disturbances 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 [59, 33, 31]. 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 solutions use many different techniques such as preemption [145] to give preference to more time-critical adaptation requirements, utility functions [44, 114, 41] to optimize component/service selection based on weights of QoS criteria, estimates of performance models [79] to select services with optimal response time, linear programming [40] to deal with different operating environments and conflicting QoS requirements, or combine linear programming with local search [13] to find configurations with minimum cost, and hybrid approaches [162] 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 [68], another approach that trades-off different qualities and provides adaptation guarantees is presented in [64] 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 [153], compared an initial version of SimCA with ActivFORMS [88], 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 [47]. Several variants have been developed for specific classes of problems, e.g. [101]. 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 [41] where the method was used to support exploring optimal controller parameters for complex industrial systems, and [70] 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 [181].

[54] 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: Handling New and Changing Requirements

In this Chapter, we present a new version of SimCA (see Chapter 5) that satisfies STO-reqs in the presence of disturbances and requirement changes. The new SimCA completes the answer to research question QI-2 “What are the appropriate models and control solutions that can be used to address STO-reqs, and deal with disturbances and requirement changes?” and research question QI-3 “What software qualities can be satisfied and what guarantees are provided with a CBSA approach that deals with STO-reqs, disturbances and requirement changes?” initially answered in Chapter 5.

In particular, new 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 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 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) [1]. Personal contribution: Stepan Shevtsov contributed 75% to the new SimCA approach, he did 100% of the technical implementation (program code, testing, etc.), performed 100% of data collection, 75% of the analysis, and 60% of writing the manuscript.
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.

6.1 Introduction

Software applications need, more than ever, being able to deal with change [136, 98]. 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 [160, 171]. 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 [43, 51, 169].

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 [163, 35, 174]. Control theory has been identified as a promising approach to design adaptation solutions with formal guarantees [25, 82, 50, 189]. 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 [67] can satisfy only one requirement at a time, while other approaches such as [68, 154] 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 [99, 130, 102] 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 [10].

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 [154], 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.

The remainder of the paper is structured as follows. Section 6.2 positions SimCA in the state-of-the-art control-theoretical approaches for building self-adaptive systems that satisfy multiple requirements. Section 6.3 elaborates on the adaptation problem that we address in this paper and illustrates it with an experi-
mental scenario. Section 6.4 presents SimCA. The formal guarantees provided by SimCA are evaluated in Section 6.5. In Section 6.6, SimCA is empirically evaluated using multiple scenarios of two cases. Finally, conclusions and directions for future research are presented in Section 6.7.

### 6.2 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. [139, 117]. 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 [50]. One of the first attempts to create such an approach is the Push-Button Methodology (PBM) [67]. 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 [68], 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 [68] and it is lacking some of the guarantees, e.g. the absence of overshooting. Finally, in [154] 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. The new 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.

### 6.3 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.*

108
6.3 Problem Definition

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 6.4 addresses these challenges.

Problem example: UUV System

We describe a UUV system (based on [151]) that we use as one of the cases to evaluate SimCA in Section 6.6 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 6.1). A scan is performed every second.

Table 6.1: 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 \( S \geq 100 \text{ km} \) should be examined within time \( t = 10 \text{ hours} \);
- **R2**: To perform the mission, a given amount of energy \( E = 5.4 \text{ MJ} \) is available;
- **R3**: Subject to \( R1 \) and \( R2 \), the accuracy of measurements should be maximized.
$R_1$ is a T-req, $R_2$ is a S-req, while $R_3$ is an O-req. We use $M_1$ to refer to normal operating mode, i.e. the UUV must satisfy the three requirements simultaneously: $M_1 = \{R_1, R_2, R_3\}$. 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 ($R_1$) using as little energy as possible of the available $E = 5.4$ MJ ($R_2$), while ensuring maximum accuracy ($R_3$), $R_1$ would become an S-req and $R_2$ 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 $M_2 = \{R_1, R_2^*\}$, where $R_2$ (S-req) changes to $R_2^*$ (O-req) defined as:

- $R_2^*$: 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 $M_3 = \{R_1, R_2, R_3, R_4\}$, where a requirement $R_4$ needs to be activated at runtime, defined as:

- $R_4$: 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 need to be guaranteed.

### 6.4 Simplex Control Adaptation* - SimCA

In this section we provide an overview of SimCA. In particular, Section 6.4 lists the element required for SimCA to build a self-adaptive system, Sections 6.4 - 6.4 describe different phases of SimCA, and Section 6.4 shows how SimCA deals with changing the set of requirements.

#### Required Elements to Apply SimCA

To build an adaptive system with SimCA, the approach requires the following elements from a software engineer:
6.4 Simplex Control Adaptation* - SimCA

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 needs 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 6.3. SimCA works in four phases that are performed during system operation, see Figure 6.1.

- 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.

Compared to the initial version of SimCA [154], 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.

1 We discuss the scope of applicability of SimCA in Section 6.6.
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 [154]. 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 $O_i$:

$$O_i(k) = \alpha_i \cdot u_i(k - 1)$$  \hspace{1cm} (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 [154].

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:

$$s_i = \frac{R_i - \Delta R_i}{t_i - \Delta t_i}$$  \hspace{1cm} (6.1)

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.

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 [154].
In SimCA, we use the following controller:

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

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 [154].

Phase III. Goal Transformation

The Goal Transformation Phase transforms all T-goals into Controller goals (C-goals), see Figure 6.2; 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 6.2: Goal Transformation phase of SimCA.](image)

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

\[
\max\{c^T x \mid Ax \leq b; x \geq 0\}
\]  \hspace{1cm} (6.2)

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 [47].

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 \( P(k) \) 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 (6.2) we refer to the linear programming literature \([47, 46, 144]\).

Knowing the vector \(x\), each T-goal is transformed into C-goal \(c_i\) as follows: \(c_i = P_i(k) \ast x\). 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 6.3. 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.

![Figure 6.3: Operation phase of SimCA (illustrated for one S- and C-goal).](image)

In particular, SimCA collects all control signals \(u_i(k)\) and the system parameters \(P(k)\) and passes these data to the simplex block. Similarly to the Goal Transformation phase, SimCA solves the system of equations (6.2) 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 (6.2) 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 (6.2) are replaced by control signals \(u_i(k)\) obtained from \(C_i\), which allows to use all the advantages provided by controllers.

### 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 6.1 with additional components, see Figure 6.4.

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

\(^2\)E.g. in the UUV scenario \(P(k)\) are the sensor parameters from Table 6.1
requirement types. Updates of requirements (goal updates) are already explained in the Identification Phase in Section 6.4.

To deal with requirement activation, the Goal Activator first transforms the requirement into a quantifiable goal (see Section 6.4) 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 6.4) 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 versa), the corresponding equality is changed to inequality in the system of equations (6.2), 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

---

3 For 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.
standard requirement deactivation procedure, while the new S-/T-req is activated according to the requirement activation procedure (see above).

6.5 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 6.5:

- 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 6.5: 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 [154]:

\[
O_i(k) = s_i \cdot (1 - p_i^k)
\] (6.3)

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 [154] we show that by analyzing the control system, we get:
6.5 Formal Evaluation of Guarantees

\[
\tilde{K} = \frac{\ln \Delta s_i}{\ln |p_i|} \quad \text{if} \quad 0 < \Delta(d) < \frac{2}{1 - p_i}
\]  

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 (setpoints) 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 6.4), 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 [154]).

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 6.4 [47, 46].

- **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 [45]. 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.

6.6 Experimental Evaluation

We empirically evaluate SimCA with two cases. First, this Section describes the experimental setting of the UUV case. Then it shows the software adaptation performed by SimCA with STO-reqs in different operating conditions. After that, we experimentally verify the guarantees and quality trade-offs provided by SimCA and test it in different scenarios. The second case with Tele Assistance System is described and evaluated in the Appendix. Finally, this Section 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 6.3 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 6.1. 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 \(\{S_1, \ldots, S_5\}\) 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 case is \(5.5 \cdot 10^6\).

\(^1\)https://people.cs.kuleuven.be/danny.weyns/software/simplex/index.htm
6.6 Experimental Evaluation

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 6.6. $\delta$ is kept at a default value: $\delta = (\max_i - \min_i) \times 0.05$.

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.

### Adaptation with STO-reqs

Figure 6.6 shows the adaptation results of SimCA on the UUV system configured according to Table 6.1 and requirements set according to UUV scenario (Section 6.3); 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 6.6) 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 6.6 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 6.6 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 ± 0.32 km, the amount of consumed energy is 5.1 MJ ± 135 J, the measurement accuracy is 89.94 ± 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 6.7. After 50 runs we got the following results: total distance scanned is 121 ± 0.28 km, the amount of consumed energy is 5.1 MJ ± 170 J, the measurement accuracy is 89.94 ± 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.

As described in Section 6.5, 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 \( \tilde{K} \) of every controller \( C_i \) depends on the pole \( p_i \) and a constant \( \Delta s_i \) chosen by the system engineer: \( \tilde{K} = \frac{\ln \Delta s_i}{\ln p_i} \). According to [82, p.85], the commonly used value of \( \Delta s \) is 0.02 (2%). Hence:

\[
\tilde{K}_{0.6} = \frac{\ln 0.02}{\ln 0.6} = 7.66 \quad \tilde{K}_{0.9} = \frac{\ln 0.02}{\ln 0.9} = 37.3
\]

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, \( \tilde{K}_{0.6} = 7.66 \times 0.44 = 3.4 \) steps and \( \tilde{K}_{0.9} = 37.3 \times 0.44 = 16.4 \) steps.
values explain why the measured scanning speed makes almost a vertical jump at $k = 100$ in Figure 6.6, while in Figure 6.7 it takes 17 adaptation steps to converge 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 6.7 it is smoother and has a much lower spike at $k = 220$ than in Figure 6.7. In this case slower reaction may be a benefit as it allows to switch less between different sensor combinations.
Figure 6.8: UUV adaptation during energy leak, $p = 0.75$. 
6.6 Experimental Evaluation

**Requirement Change Scenario: Energy Leak**

Figure 6.8 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 6.3 and 6.6) 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 $M_2 = R_1, R_2^*$. Summarizing, 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 6.8 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 6.8. 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 6.9. 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
Figure 6.9: New requirement activated at runtime, $p = 0.9$. 

![Graphs showing Energy Consumption, Scanning Speed, Fail Rate, and Sensor usage over time.](image)

- **Energy Consumption**
- **Scanning Speed**
- **Fail Rate**
- **Sensor usage**

*Figure 6.9: New requirement activated at runtime, $p = 0.9.*
6.6 Experimental Evaluation

Figure 6.10: UUV adaptation with 2 sensor panels, $p = 0.6$.

UUV enters the mode $M3 = R1, R2, R3, R4$, with $R4$: The average failure rate should be $F \leq 0.02\%$.

Figure 6.9 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 $R1$ at $k = 100$ and 170; we also shut down $S4$ at $k = 215$ and increases the energy consumption of $S1$ 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 6.9. For example, when $S4$ stops working, the UUV is forced to use sensor $S5$ and more of $S3$.

**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 char-
characteristics similar to those in Table 6.1. 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 6.10 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 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 [59, 33, 31]. However, the deployment of SimCA in a real-world setting is required to confirm the obtained results in practice.

6.7 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 [172]. 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 6.11.

![Figure 6.11: TAS workflow.](image)

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 6.2 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 $TAS_{FR} = S_{1FR} + AS_{1FR} = 0.36\%$, 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:

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

128
6.7 Conclusions

- 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 \( M1 = \{ R1, R2, R3 \} \). 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 [172]. The results of SimCA applied to a TAS scenario are shown in Figure 6.12. 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

![Figure 6.12: SimCA on TAS scenario.](image_url)
$k = 295$ and the system enters mode $M_2 = \{R_2, R_3\}$, where TAS uses services $S_5$ and $AS_3$, 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.
Stage II. Handling Uncertainties

This part of the dissertation describes the research stage II in details and how we solved the research problem \textit{RPII}. It includes three chapters:

- Chapter 7 provides an overview of research stage II. It starts by defining uncertainty and its types. Then, the chapter introduces the research questions tackled and the research methods being used during stage II.

- The results of a descriptive literature review that studies automated control-theoretical approaches to realize self-adaptive software are given in Chapter 8. The chapter analyzes problems tackled by existing automated CBSA approaches, finds pros and cons of these approaches, and identifies gaps in the existing research on this topic.

- Chapter 9 presents SimCA* that builds upon SimCA and deals with different types of uncertainties. The chapter describes new components of SimCA* that deal with uncertainty in software parameters, addition or removal of requirements at runtime and software component interactions. This chapter also analyses qualities and guarantees achieved by systems equipped with SimCA* using two software applications.
Chapter 7

Overview of Research Stage II

This Chapter gives an overview of research stage II. The necessary background on uncertainty and its types is given in Section 7.1. The research problem and research questions studied during stage II are described in Section 7.2; Section 7.3 outlines the research methods being used to tackle this problem.

7.1 Background: Uncertainty

According to [169], software systems have to face four main types of uncertainty: (i) the uncertainty that comes from the software itself, (ii) the uncertainty that stems from the system goals, (iii) uncertainty introduced by the execution environment and, (iv) uncertainty due to human-related aspects. In this thesis, we study three of these types of uncertainties:

- We deal with uncertainty in system parameters and component interactions that come from the software itself. Here, uncertainty in component interactions means that an adaptation action performed by one system component may affect adaptations performed by other components.
- We handle uncertainty in requirements that may affect the qualities provided by the system. In our case, we consider only anticipated uncertainty in requirements, meaning that the system goals that represent non-functional (or quality) requirements (response time, failure rate) can be activated, deactivated, or their values may be adjusted at runtime based on conditions that are defined before deployment, but that can only be resolved during operation.
- We deal with environment uncertainty. Uncertainty in the environment means that the system may be affected by external disturbances from different sources, such as noise, signal interference, etc.

Human-related uncertainty (like the presence of humans in the loop, or multiple ownership of software elements) is out of scope of this work.

7.2 Research Problem

Based on research problem RPII “To devise a reusable CBSA solution that satisfies multiple stakeholder requirements in the presence of different types of uncertainty, and provides guarantees on the adaptation results”, we identified two research questions.
Since research stage I, when we performed the systematic literature review (see Chapter 4), the research in automated CBSA evolved and a number of new approaches appeared in parallel with SimCA. Therefore, with research question QII-1 we want to study those automated CBSA approaches in details and see whether any of them deals with RPII or its parts:

**QII-1:** “What types of requirements, uncertainties and guarantees are addressed by existing automated control-theoretical approaches that realize self-adaptive software?”

SimCA developed at research stage I only deals with some types of uncertainty (disturbances and requirement changes). With research question QII-2, we would like to tackle other uncertainties, such as addition or removal of requirements during operation, runtime changes of software parameters and interactions between software components:

**QII-2:** “How to tackle different types of uncertainty with an automated CBSA solution that deals with multiple requirements and provides adaptation guarantees?”

### 7.3 Research Method

In this Section we provide a detailed description of the methods used to conduct the research in stage II, see Figure 7.1. In particular, we discuss descriptive literature review and informal exploratory case studies in the following subsections.

---

**Figure 7.1:** The detailed scheme of research stage II
Overview of Research Stage II

Descriptive Literature Review

A descriptive literature review (also known as mapping review) is an informal research method that helps to get a quick overview of research that has been conducted on a chosen topic [137].

A descriptive literature review typically consists of four steps, see Figure 7.2. The review may have different goals, such as allocating gaps in existing research, analyzing research articles that tackle a specific problem, classification and characterization of existing solutions, patterns or frameworks in a certain area, and others [113]. So the first step of a descriptive literature review is to define its goals. The second step is to select research publications according to the review goals. This process is rather informal, so it may include different searching techniques or just be based on researcher’s experience in the selected topic. The assessment of inclusion and exclusion criteria or bias is not required for a descriptive literature review. The third step of the review is to analyze the selected studies according to the previously set goals. Finally, the review results are used as a background section in a research article or published as a separate research work.

Our descriptive literature review focused on automated control-theoretical approaches to realize software adaptation that were developed after research stage I, i.e., after we conducted the systematic literature review of the CBSA research field. The review showed that existing automated CBSA solutions tackle only a small part of uncertainties (QII-1), typically focusing on disturbance rejection and dealing with changes of requirement values. The review results provided a background and justification for the development of SimCA*, see Figure 7.1.

The overview of conducted descriptive literature review with detailed results is presented in Chapter 8.

Informal Exploratory Case Studies

In order to answer the research question QII-2, we built upon our previous approach SimCA (see Chapter 6) and the knowledge gained from the descriptive literature review (see above). As a result, we introduced SimCA*, a new control-theoretical approach to realize self-adaptive systems, see Figure 7.1. Based on our successful experience with using informal exploratory case studies to validate SimCA, we decided to use this research method to validate SimCA* and study its effectiveness as well. The informal exploratory case study method is described in details in Section 3.3.

Similarly to research stage I, we used informal exploratory case studies to understand and characterize the behavior of software systems equipped with SimCA*. Our approach was first verified on a typical system with strict requirements, i.e., the UUV system performing surveillance missions. Then, in order to observe how
SimCA* deals with uncertainty in interactions between software components, it was tested on a simulated Internet of Things network (DeltaIoT system [164]). In order to complete the answer to QII-2, we analyzed software qualities and guarantees obtained by the developed CBSA approach.

Informal exploratory case studies of SimCA* with both the UUV system and the DeltaIoT system are available in Chapter 9.
Chapter 8

Overview of Control-Theoretical Approaches to Realize Self-adaptation

In this Chapter, we report the outcomes of a descriptive literature review that investigated automated control-theoretical approaches to realize self-adaptive software. In total, we extracted data from 7 research papers.

The main goal of the literature review was to answer the research question QII-1 “What types of requirements, uncertainties and guarantees are addressed by existing automated control-theoretical approaches that realize self-adaptive software?”

The descriptive literature review:

• Studied research problems tackled by existing automated CBSA approaches;
• Classified automated CBSA approaches, found their pros and cons;
• Listed a number of gaps in the existing automated CBSA research and discussed challenges of this research filed.

Reflecting upon results of the literature review, we found out that existing automated CBSA solutions can deal with different types of requirements and provide adaptation guarantees. However, they take into consideration only few types of uncertainties, typically focusing on disturbance rejection and dealing with changes of requirement values at runtime. This fact justified the development of SimCA®, our own approach to deal with different types of uncertainties, see Chapter 9.

This Chapter is a copy of an accepted chapter from an upcoming book “Engineering Adaptive Software Systems” [156]. Personal contribution: Stepan Shevtsov performed 90% of the initial analysis of the state of the art in control-theoretical software adaptation and contributed 60% to writing the manuscript.
Self-adaptation of Software using Automatically Generated Control-theoretical Solutions

Abstract

Control theory has contributed a set of foundational techniques to handle “change” at runtime in software applications. These techniques however have fundamental limitations as well: (i) they require the development and understanding of mathematical models; (ii) synthesizing solutions is often done on a per-problem basis, discouraging flexibility and generality. Software engineering, as a discipline, has always aimed at finding reusable and modular solutions. The combination of the desire to apply formally grounded control-theoretical principles and reuse existing solutions has motivated research on the topic of automatically generated control solutions. This research aims at designing control strategies in an automated way from data that qualifies the given problem at hand. This chapter provides an overview of the research topic of automatically generated control-theoretical solutions, explaining the key research contributions and paving the way for future research.

8.1 Introduction

Software applications need, more than ever, to be able to deal with “change” [136, 98]. Software needs to be continuously available, which in turns requires that developers treat change as a first class concern in the complete lifecycle of the application development, operation, and maintenance. Software applications are nowadays expected to deal seamlessly with different types of change, such as resource fluctuations [125], component failures [150], requirement modifications [158, 11], different user-preferences [149], and much more [146, 83, 17, 5, 3, 38]. Often, these changes are not predictable at design time, requiring software to execute with incomplete knowledge and face new challenges during operation [160, 171]. Consequently, software engineering researchers are experimenting with new solutions that can handle change at runtime without incurring into penalties, slowdown, and downtime. Generally speaking, the software built to deal with change is often called “self-adaptive” [43, 51, 169], for the ability to modify its own behavior and adapt to the current execution conditions.

Continuous- and discrete-time control theory

Continuous- and discrete-time control theory has been identified as a promising approach to design self-adaptive software [25, 82, 50, 189]. However, the wide adoption of control-theoretical solutions in the design of self-adaptive systems has been limited by a number of factors.

First and foremost, continuous- and discrete-time control solutions often require a “physical” model of the object to be controlled. In the case of low-level

---

1In this chapter, we restrict ourselves to continuous- and discrete-time control [16, 179]. Discrete event systems are out of our scope.
resources—such as CPU, memory, and network bandwidth—researchers have proposed models that attempt to capture the phenomena of interest [4, 53, 190] with a precision sufficient to perform adaptation. However, it is very difficult to extract control-theoretical (i.e., equation-based) models for the behavior of software applications. This has been one of the main reasons why several researchers have argued that applying control theory to adapt the higher-level software elements is a more complex problem [67, 8, 24]. Other reasons are the diversity and interplay of requirements, and the need for instrumenting software to obtain measurements from sensors and enacting the system through actuators [28, 85]. Second, the models often become complicated, calling for elaborate solutions from the mathematical perspective. Finally, since appropriate and accurate models are so difficult to write, existing control-based approaches are often tailored for a particular problem, while software engineers usually aim for reusable solutions. These observations have been recently confirmed by a systematic study on control-theoretical software adaptation, highlighting the shortcomings of the existing ad hoc control-theoretical solutions [2].

As a response to these shortcomings, researchers aimed at automatically generating control solutions. These solutions are general enough to tackle a variety of problems, trading-off the optimality that could be reached by tailored solutions. The code for these general solutions can be automatically generated based on observations and data from the software application that should be controlled. Simple linear models describing the software behavior are automatically extracted from the data and used—at runtime—to synthesize a control solution. This chapter gives an overview of the state of the art of the research in automatically generated control strategies for software applications and outlines promising paths for future work.

The remainder of this chapter is structured as follows. Section 8.2 provides a brief background on automatically generated control-theoretical adaptation of software. In Section 8.3, we delve into details discussing the differences among the proposed solutions. Finally, Section 8.4 outlines a number of challenges for future research and Section 8.5 draws some conclusions.

8.2 Background

This section explains the basic principle behind automatically generated control-theoretical solutions and its use for self-adaptation.

The overall objective of automatically generated control-theoretical adaptation is the simplification of the software design process. The aim of these strategies is to provide the software engineer with the advantages of a control-theoretical design, without the need for in-depth control expertise. The main advantage of control-theoretical solutions is the presence of formal guarantees [69]. If mastered correctly, the use of the knowledge coming from control theory allows for certified and verifiable solutions, where desired properties can be guaranteed by design. For example, with control theory it is possible to precisely calculate the amount
Overview of Control-Theoretical Approaches to Realize Self-adaptation

Figure 8.1: A typical control-theoretical feedback loop

The feedback loop shown in Figure 8.1 is used in self-adaptive software systems. Reading the figure from left to right, the *Goal* represents a particular level of software quality that should be achieved by self-adaptation. The Goal is often specified as a setpoint, i.e., a certain value of a non-functional requirement, such as a specific service failure rate or response time. Using the setpoint and the *Measured Output* value for the same software quality, an *Error* is calculated as *Setpoint* – *MeasuredOutput*, where the -1 block indicates that the Measured Output value should be subtracted. The *Feedback Controller* uses the Error in order to compute the *Control Signal*, a value or a vector of values that affect the *Software System*. If designed correctly, the Control Signal will result in a Measured Output that is equal or very close to the Goal value. The *Disturbances*, such as changing availability of resources or component failures, affect the software behavior at runtime. So one of the main purposes of control strategies is to neglect the effect of Disturbances on the system.

Historically, many manually generated control strategies used the typical feedback loop shown in Figure 8.1. The automated strategies have two main differences from these solutions. First, the automated strategies require certain conditions to be satisfied and the availability of specific software functions:

- The developer that wants to generate and use the control strategy should have access to the software system, which should be working and on which experiments should be done and data must be collected – the data is used in an automated way to build a model of the software that can be used for control purposes;
- The developer should be able to qualify, quantify, and measure the requirements that must be satisfied on the system – these requirements are then translated into goals and objectives that the controller will try to achieve;
- The developer must provide access to a set of sensors that get reliable data about the quantifiable objectives (e.g., measure the response times of a cloud application);
- The developer must provide access to a set of actuators (tunable parameters of the system) that can be used during runtime to modify the behavior of the software application (e.g., the percentage of rejected requests, or different implementations of the same functionality).
Second, the Feedback Controller is created automatically. Namely, the automated solution starts by running experiments on the software application, changing the values of the actuators according to predefined patterns and measuring the values of the goals in the tested configurations. With this data, the solution generates a mathematical model of the software using system identification [116]. Finally, this model is used to synthesize a controller that provides guarantees on certain system properties. The controller – synthesized in form of equations and subsequently in form of a code block – adapts the behavior of the software changing the values of the actuators to achieve the given goals. The resulting controller is often tunable – some parameters have default values, that can be changed to alter the behavior of the controller itself. For example, parameters can be used to exploit the trade-off between robustness to disturbances and speed of convergence. The software engineer can select these parameters based on experience and on the specific execution conditions.

8.3 Automated Control-Theoretical Software Adaptation

This section outlines the research progress in self-adaptation of software using automatically generated control-theoretical solutions. We discuss five different research problems that have been explored. Figure 8.2 gives an overview of the research steps and shows representative approaches for each step. The arrows in the Figure show the contribution of each step/approach to the following efforts.

![Figure 8.2: Research in automated control-theoretical software adaptation: progress steps (left) and approaches (right).](image)

The initial research was primarily targeting the automation of a control solution development. Based on prior experience with control of software applications, some generalization arose and led to the introduction of the Push-Button Methodology (PBM) [67]. At the same time, a similar method called Brownout [102]

---

1 Other model synthesis techniques can be used to produce system model. But historically, automated approaches used system identification as it is fast and approximates software well enough for controllers to work.
was applied in a specific software domain, cloud applications. The next clear research goal has been the extension of automated methodologies to support *multiple adaptation goals* simultaneously, e.g., to achieve a specific performance level and minimize cost at the same time. The first proposed extension has been the Automated Multi-objective Control of Software (AMOCS) approach [68], followed by the Simplex Control Adaptation (SimCA) [154]. SimCA tackled the problem of multi-objective adaptation by combining controllers with the simplex optimization algorithm in a hierarchical structure. Then, the new version of SimCA [1] introduced components that adjust the adaptation mechanism at runtime, to deal with new types of goals and changes in the set of adaptation goals (e.g., adding a new goal, removing a goal). Finally, the use of *Model Predictive Control* (MPC) was investigated. In this approach, the controller acts based on the current feedback from the software, but uses the model of its own behavior to predict the software evolution. The fully automated MPC-based approach is called Automated Multi-objective Control of Software with Multiple Actuators (AMOCS-MA) [120].

The main properties of all automated control-theoretical adaptation approaches are listed in Table 8.1, these approaches will be discussed in details in the following sections.

**Table 8.1: Automated Control-Theoretical Adaptation Approaches**

<table>
<thead>
<tr>
<th>Approach</th>
<th>Goals</th>
<th>Main Pros</th>
<th>Main Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brownout, PBM</td>
<td>1-setpoint</td>
<td>Automation, guarantees</td>
<td>Handles only one goal</td>
</tr>
<tr>
<td>AMOCS</td>
<td>n-setpoint, 1-optimiz.</td>
<td>Multiple goals and prioritization</td>
<td>Sub-optimal adaptation decisions</td>
</tr>
<tr>
<td>SimCA</td>
<td>n-setpoint, 1-optimi.</td>
<td>Guarantees + optimality</td>
<td>Setpoints, needs knowledge about some of the system parameters</td>
</tr>
<tr>
<td>new SimCA</td>
<td>n-setpoint, n-threshold, 1-optimiz.</td>
<td>Handles new types of goals and goal changes at runtime</td>
<td>Needs knowledge about some of the system parameters</td>
</tr>
<tr>
<td>AMOCS-MA</td>
<td>n-setpoint, 1-optimiz.</td>
<td>Guarantees + optimality, does not need system knowledge, flexible computation time</td>
<td>Sensitive to disturbances and model inaccuracies</td>
</tr>
</tbody>
</table>

**Automation of Control System Development**

Control-theoretical approaches were first used in software adaptation more than a decade ago [5, 3, 38]. However, most of these approaches aim to solve a specific problem at hand. Therefore, new problems would require modifications or even replacement of a control system, which in turn requires expertise in control theory, extra resources and effort. To overcome this concern, researchers have studied the ways to automate the entire process of control system development from the
model synthesis to the formal analysis of guarantees. This became the first step of research on applying automatically generated control-theoretical solutions in software adaptation.

The representative of the first step of research are Brownout [102] and PBM [67]. Both these approaches are based on the same underlying principles (creating a first order model from data and controlling that first order model using pole placement). Brownout is applied to the more confined domain of cloud computing applications, and is tailored to the specific problem of capacity shortages. Because of this, Brownout achieves – on its own problem – better performance than the application of the PBM controller without any modifications. We provide details on both of these approaches below.

**Brownout**

The main idea behind Brownout [102, 123] is to apply the principles of graceful degradation to cloud applications using control theory. Cloud applications behave according to the request-response paradigm, with clients issuing requests and a certain number of replicas of the same application providing the according responses. When producing the response to the user requests, it is often possible to identify a part of the response that is the mandatory to display and a part of the response that would provide a better user experience and increased revenues, but not mandatory. In the case of a travel agency website, the mandatory part of the response is the flight search, while additional optional information are car rental locations and hotel suggestions. Clearly, the application owner wants to provide the additional information, but not at the expense of losing a customer. Brownout divides the response into the two mentioned parts and measures the response time to determine how much percentages of the optional content should be served. This percentage is called the *dimmer* value. The goal of brownout is to have as big dimmer as possible, i.e., to show as much optional content as possible, without penalizing response times.

Brownout assumes that the cloud application behaves according to a simple first-order linear model, where the value of the 95th percentile of the response time \( \tau_{95} \) varies depending on the dimmer value as follows:

\[
\tau_{95}(k) = \alpha \theta(k - 1) + \delta \tau_{95}(k),
\]

where \( \theta(k) \) is the dimmer value; \( \alpha(k - 1) \) is a time-varying coefficient that depends on the computing platform and can be estimated; \( \delta \tau_{95}(k) \) is a disturbance, interfering with the nominal system’s behavior; \( k \) is the discrete time instance.

Based on the model (8.1), the following controller is then synthesized using loop shaping [16]:

\[
\theta^*(k) = \theta(k - 1) + \frac{1 - p_b}{\hat{\alpha}(k)} \cdot e_{95}(k)
\]

where \( \hat{\alpha}(k) \) is an estimate of \( \alpha(k) \) obtained with a Recursive Least Square (RLS) filter; \( p_b \) is a controller parameter called pole; \( e_{95}(k) \) is the error between the desired 95th percentile of the response time \( \bar{\tau}_{95}(k) \) and the actual value. The
pole $p_b$ can be used to trade the speed of controller convergence for robustness to model perturbations. The analysis of the brownout closed loop allows to prove a number of properties, such as system stability and zero steady-state error. However, this proof is subject to how well the model (8.1) approximates the behavior of the cloud application.

Brownout uses a single actuator (the dimmer value) to achieve a single goal, specified in terms of a setpoint for the response times statistic. The control strategy in Brownout can be greatly improved and many follow-ups were devised. For example, an event-based version of the brownout paradigm [52] explores a similar cloud problem, but controls the server queue length. Furthermore, extensions that include brownout load balancing were considered [57, 103]. They demonstrate that state-of-the-art load balancers which use response times as a measure for determining where to send requests do not work with brownout-aware applications. This is a natural limitation as the brownout controller can satisfy only a single goal and therefore cannot form a multi-objective control strategy with other controllers.

Brownout was designed specifically for cloud applications, so strictly speaking, it is not a generally applicable solution. However, it is important to include Brownout in this work as it became the first building block for development of automated control-theoretical adaptation. The generally applicable Push-Button Methodology, discussed in the following section, is based on the same principles and shares many elements with Brownout.

**Push-Button Methodology**

The PBM methodology [67] works in a way similar to Brownout, but goes beyond a single goal and a single actuator. Also, it introduces the idea of identifying the model online. Unlike in Brownout, where model is pre-determined, PBM builds a model directly from the data received by running experiments on the software and produces a controller for this model. Figure 8.3 shows the two phases of the methodology: model building and controlling.

![Figure 8.3: The two operational phases of PBM.](image)

The input required by PBM from a software engineer is a method to set the actuator value and a method to collect measurements about the system goal. Based on this input, PBM first produces a linear model $\mathcal{M}$ of the software:

$$\mathcal{M} : y(k) = \alpha(k - 1) \cdot u(k - 1)$$

(8.3)
8.3 Automated Control-Theoretical Software Adaptation

Where the input $u$ is the value of the actuator; the output $y$ is the effect of the actuator on the goal; the parameter $\alpha$ is a time-varying coefficient that is determined during model building by feeding different input values as $u$ and measuring the resulting outputs $y$; and $k$ is a discrete time instance.

After the model building, the controller synthesis phase automatically generates a Proportional-Integral controller $C$ that works on the model $M$ and adapts the software.

$$C : u(k) = u(k-1) + \frac{1-p_b}{\alpha} \cdot e(k)$$  (8.4)

The controller has one parameter, $p_b$, that has the same role that it had in the Brownout controller. More guidelines on how to tune the controller parameter $p_b$ are available in [67].

To address model inaccuracies and small perturbations during software operation, the value of $\alpha$ is updated at runtime. In case of critical changes (e.g., a software component failure), PBM restarts the model building phase and regenerates the controller.

**Adaptation with Goal Prioritization**

In order to automatically create control solutions for more practical problems, researches have studied the ways to address multiple adaptation goals simultaneously. The first automated approach that offered control-based multi-objective software adaptation was AMOCS [68]. This approach extends the methodology behind PBM to use multiple actuators and multiple controllers in a cascaded structure, see Figure 8.4.

![Figure 8.4: A self-adaptive software with AMOCS (for 2 goals).](image)

AMOCS works as follows. The set of available actuators $A = \{a_1, \ldots, a_m\}$ is partitioned to reach the set of goals $G = \{g_1, \ldots, g_n\}$, where $m \geq n$, i.e., the system should have more actuators than goals. The goals are added into the set $G$ according to their priority order, forming the chain $< g_1, g_2, \ldots g_n >$, where $g_1$ is the most important goal and $g_n$ is the least important one. All goals, except the last one, are specified as setpoint values to be achieved by the adaptation. The last goal $g_n$ is always the optimization of a specific value (e.g., maximization of profit, minimization of cost). $A_i$ denotes the subset of actuators used to achieve the goal $g_i$. AMOCS assumes that every actuator is used:

$$\bigcup_{i \in \{1 \ldots n\}} A_i = A,$$  (8.5)
and each actuator is assigned to a single goal only:

$$\forall i, j \in \{1 \ldots n\}, i \neq j \implies A_i \cap A_j = \emptyset,$$

(8.6)

A first instance of PBM controller $C_1$, see (8.4) for a controller description, is then used to translate the discrete set of configurations of all the actuators $A_1$ related to the first goal $g_1$ into a single configuration that satisfies this goal. This configuration is then sent in the form of control signal $k_1$ to the software system and to the second instance of PBM controller $C_2$, that tries to achieve the second goal $g_2$ with the available actuators $A_2$ and operating conditions. The resulting configuration is send to software as control signal $k_2$. If goals $g_1$ and $g_2$ are not related, the control signal $k_1$ will still be received by controller $C_2$, but it will not affect the reachability of the goal $g_2$. In this controller chain, only the first goal is guaranteed to be stable, while the stability of the others depend on the disturbances and on the control values set by the previous controllers in the chain. In other words, the goal $g_2$ is guaranteed to be reached only if control signal $k_1$ allows to reach it. The last optimization requirement is reached to the best of the chain ability, hence there is no guarantee for the solution optimality. Despite the lack of formal guarantees, the experiments with AMOCS show that the chain of controllers behaves well in a variety of different scenarios and can successfully handle multiple goals of a setpoint type.

Adaptation with Guaranteed Optimality

Guided by the need for stronger adaptation guarantees in systems with multiple goals, the research explored new ways to automatically build the control system. The approach resulting from these efforts is called Simplex Control Adaptation (SimCA) [154]. SimCA combines PBM with the simplex optimization method, utilizing the advantages of both approaches. SimCA finds 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 PI-controllers that operate on these models and independently compute control signals for each of the goals, and combines controller outputs using the simplex method to adapt the system. Figure 8.5 schematically shows the primary building blocks of SimCA.

![Figure 8.5: A self-adaptive software with SimCA.](image-url)
SimCA builds a self-adaptive system in three phases executed during system operation:

1. In the Identification phase, \( n \) linear models of the controlled system are built. SimCA uses multiple instances of the PBM model \( M \), where each model \( M_i, i \in [1, n] \), is responsible for one goal \( s_i \). Similar to PBM, each model is automatically learned at runtime by running the experiments on the software (see Section 8.3 for details). As in PBM, the model \( M_i \) automatically adjusts at runtime according to changes in the system behaviour.

2. In the Controller Synthesis phase, SimCA constructs a set of \( n \) controllers; each controller \( C_i \) is responsible for the \( i \)-th goal. \( C_i \) 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 coefficient \( \alpha_i \), parameter pole \( p_i \) and the error \( e_i(k - 1) \), with \( e_i = s_i - O_i \). Similar to PBM, \( p_i \) is used to tune the controllers and trade-off different system properties.

\[
\begin{align*}
    u_i(k) &= u_i(k - 1) + \frac{1 - p_i}{\alpha_i} \cdot e_i(k - 1) \quad (C_i)
\end{align*}
\]

3. In the Operation 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 \( u_i(k) \). SimCA combines the signals \( u_i(k) \) from all the controllers and uses the simplex method to calculate 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:

\[
\max \{ c^T x \mid Ax \leq b; x \geq 0 \} \quad (8.7)
\]

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 [47].

In SimCA each equation, except the last one, represents a goal \( s_i \) 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 \( x \).

Note that SimCA uses a simplex variant with equalities \((A x = b)\) in order to prevent simplex from changing the effect of control signal \( u_i(k) \) on the output signal \( O_i(k) \). Instead, simplex is responsible for seamless translation of control signals \( u_i(k) \) to actuation signal \( u_{sx} \). This allows to provide the entire set of control-theoretical guarantees, including stability, absence of overshoot, tunable settling time and robustness to disturbances. A major advantage of SimCA over approaches from the previous research steps is that
simplex guarantees solution optimality, meaning that all the system goals are guaranteed to be achieved. An interested reader may refer to [154] for further details.

A follow-up work [153] compares SimCA with an architecture-based ActivFORMS approach using a simulated service-based system. The study shows that both approaches can deal with multiple goals and provide guaranteed solution optimality. However, SimCA achieves better results in the presence of runtime changes as it does not rely on data verified at design time. Except optimality, the two adaptation approaches offer different guarantees. The design of SimCA adaptation mechanism allows to formally prove the properties of underlying system and guarantee that they will hold at runtime independent of the system parameters. ActivFORMS, on the other hand, can guarantee the functional correctness of the implementation of the adaptation algorithm, such as the absence of erroneous states and correct interaction between adaptation components.

Adaptation with New and Changing Goals

One interesting research line for automated methodologies and for control methodologies in general is the selection and support of types of adaptation goals. The previously developed automated approaches had two major drawbacks. First, they addressed goals specified either in the form of particular setpoint values to be achieved by the system (S-goal) or values to be optimized (O-goal), while many software systems need to address a threshold goal that keeps a value above/below a threshold (T-goal). A typical example is limiting the response time of a web server. Approaches such as described in [99, 130, 102] solve this problem either by optimizing the response time (O-goal) or by defining a setpoint for response time that the controller should guarantee (S-goal), when the actual requirement is to keep response time lower than a certain threshold. Second, the previously developed approaches did not provide support for changing the set of system requirements during operation, which requires on the fly adjusting, activation and deactivation of adaptation goals. 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 order to address the two mentioned concerns, the SimCA approach (see Section 8.3) was reworked and upgraded [1]. Compared to original SimCA, the new approach includes an additional Goal Transformation phase (Figure 8.6) and the necessary mechanisms to support changing system requirements by activating/deactivating goals (Figure 8.7).

The Goal Transformation Phase of SimCA is performed between the Controller Synthesis and Operation phases. The purpose of this phase is to transform T-goals into goals that can be controlled by the original SimCA controller \((C_i)\). As such, the approach uses simplex, where each equation in the system (8.7), except the last one, represents an S-goal or T-goal to be satisfied (see Figure 8.6). Equalities are used for S-goals, while inequalities are used for T-goals. The last equation ensures that the system selects a valid solution, the vector \(x\), by the means of constraints, e.g. \(x \geq 0\). The goal of simplex is to find such vector \(x\) that satisfies all system
goals; the details of how simplex finds such a solution can be found in the linear
programming literature [47]. Knowing the vector $x$, each T-goal is transformed
into a controller goal (C-goal) $c_i$ as follows: $c_i = P_i(k) * x$. The resulting 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. All the C-goals and the original S-goals are then
used by controllers ($C_i$) in the usual Operation phase described in Section 8.3.

![Figure 8.6: Goal Transformation phase of SimCA.](image)

In order to address the changing system requirements, SimCA is equipped with
a Requirement Monitor, Goal Activator and Goal Deactivator components, see
Figure 8.7. The Requirement Monitor triggers the corresponding adaptation com-
ponent after any system requirement is changed. The Goal Activator first reads
the relevant parameters $P$ related to the activated goal. Then, in case of O-goal
activation, it inserts $P$ into the objective function $c^T$ of simplex, performs a Goal
Transformation (described above) and proceeds to standard Operation phase. In
case of S- or T-goal, the Goal Activator triggers a standard Identification phase for
the new goal, which is followed by Controller Synthesis, Goal Transformation and
Operation. The Goal Deactivator removes the according elements of the adapta-
tion mechanism. Namely, when an S- or T-goal is deactivated, the corresponding
controller is removed together with the equation responsible for the goal being de-
activated. When an O-req is deactivated, the corresponding variables are removed
from the objective function $c^T$ of simplex. After that, 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.

**Automated Model Predictive Control**

The scope of applicability of the first multi-objective control solutions is limited
in different ways. For example, SimCA cannot prioritize goals or use infinite sets
of values for the actuators, while AMOCS produces sub-optimal solutions. **To elimi-
nate these limitations, researchers have studied the application of automated model
predictive control (MPC) – a technique based on the optimization of a cost
function and on the prediction of a future outcome of the adaptation. Generally,
in control theory, MPC is considered particularly well suited for multi-objective
problems with optimization, because all the inter-dependencies between actuators
and goals are taken into account simultaneously, achieving a truly optimal solution.
The first research effort that identifies automated MPC as a potential multi-objective control strategy for self-adaptive systems is [9]. However, it lacks details and does not provide any analysis of guarantees. In the same research line – again for a specific problem, but with a general overlook – CobRA [10] provides a framework to reason about MPC and its application to computing systems. Although the model in CobRA has to be generated manually and fed to the system, the solution of the MPC problem is general with respect to the involved quantities. The paper only provides an example of the framework application, which also requires extensive manual tuning in order to tailor the equations to a specific problem. Although formal guarantees are not discussed in CobRA, it is possible to prove that they hold to the extent that the model allows. PLA [131, 130] is based on similar principles that CobRA. It uses a model of the environment and of the software to determine the best strategy to be followed using a model checker with the ability of looking into the future expectations for the system. CobRA and PLA have been compared [132] showing similar results, but a different runtime behavior. The authors conclude that the concrete approach should be picked based on the problem at hand. For example, CobRA suits more for continuous inputs, while PLA works better with discrete control.

Finally, a fully automated model predictive control strategy was developed as a part of AMOCS-MA approach [120]. Similar to other automated solutions, AMOCS-MA starts with a model building phase. The following model $S$ is synthesized:

$$S = \begin{cases} x(k+1) = A \cdot x(k) + B \cdot \Delta a(k) \\ O(k) = C \cdot x(k) \end{cases} \quad (8.8)$$

where $k$ is a discrete time instance; $O(k)$ is the vector of all system outputs at time $k$; $\Delta a(k)$ is the control signal containing values of all actuators; $x(k)$ is the current system state; $x(k+1)$ is the next system state; $A$, $B$ and $C$ are the matrices of coefficients obtained with model learning by running experiments on the software at runtime. One of the AMOCS-MA advantages is that it reduces the
8.3 Automated Control-Theoretical Software Adaptation

**Figure 8.8:** A self-adaptive system with AMOCS-MA

The model learning time by using special input signals in the model building phase, see details in [120]. As in other automated approaches, the model is updated according to runtime changes that appear in the software system.

The model $S$ is used by an MPC controller to minimize the following cost function, which handles all $S$-goals and $O$-goals:

$$
\text{Minimize } \Delta a(k+i-1), \text{ with } i = 1 \ldots L \text{ in:}
$$

$$
L \sum_{i=1}^{L} \left( \sum_{j=1}^{p} q_j \cdot |O_j(k+i) - g_j(k+i)|^2 + \sum_{l=1}^{m} r_l \cdot \Delta a_l(k+i-1)^2 \right)
$$

Subject to: model $S$ (8.8) and additional $\Delta a(k)$ constraints (see [120])

where $k$ is a discrete time instance; $L$ is the number of discrete time instances in future used for predicting software behavior; $p$ is the number of goals; $q_j$ is the weight of goal $j$ (allows goal prioritization); $O_j(k+i)$ is the predicted measured output of goal $j$ at the $i$-th step in future; $g_j(k+i)$ is the value of goal $j$ at the $i$-th step in future (this value is constant if goals do not change at runtime); $m$ is the number of actuators; $r_l$ is the weight of actuator $l$ (allows actuator prioritization); $\Delta a_l(k+i-1)$ is the predicted change in the value of actuator $l$ at the $i-1$-th step in future.

As the controller depends on the model (8.8), it requires information about the system state $x(k)$. However, it is problematic to measure the system state directly, so it is estimated instead. To accomplish this, AMOCS-MA uses a Kalman Filter that computes an estimate $\hat{x}(k)$ of the state $x(k)$ based on the previous control signal $\Delta a(k-1)$, the measured outputs $O_j(k)$, prediction error and a number of other parameters.

Using the estimate $\hat{x}(k)$, the MPC controllers solves (8.9) and produces an optimal plan of control actions for the future $i$ steps: $\Delta a(k+i-1)$, with $i = 1 \ldots L$. The plan $\Delta a(k+i-1)$ contains particular values of all actuators at time instance $(k+i-1)$. However, AMOCS-MA uses only the first action of the plan, i.e. $\Delta a(k)$ is applied to software, see Figure 8.8.

The controller (8.9) guarantees stability, zero steady state error, and minimal settling time by design. It also guarantees the optimality of a cost function specified by the user. This function has tunable weights for the system goals $q_j$ and
actuators $r_t$, allowing to trade-off different system properties, e.g. to prioritize response time over cost.

### 8.4 Challenges

The analysis of automated control-theoretical adaptation solutions showed the use of various controllers, from hierarchical adaptive PI-control (SimCA) to model predictive control (AMOCS-MA). However, most of these approaches use the PBM model (8.3) or its variations. Indeed, one of the key points behind this line of research is the difficulty in finding generic models that describe software applications and their behavior. Although the usual software models – architectural models, UML descriptions – are a very good reference to understand how the control code interfaces with the rest of the software application, they are not suitable for the control design process. To design a controller, there is usually a need to understand how the quantities that should be controlled are influenced by the actuators that one has available. Depending on the modeling effort that the software engineer is willing to do, the control strategies can be more or less effective:

- **PLA [131, 130] and Brownout [102]**, for example, use explicit modeling of both the software behavior and the environment. Explicit modeling goes a long way for improving the performance of the control strategy, that can be perfectly tailored for a new scenario using the given knowledge. Generally speaking, when an explicit model is available, the spectrum of results that it is possible to obtain is much wider, opening up possibilities and allowing for more precise results.

- **SimCA [154, 1]** lifts some of the requirements on the modeling side. While no explicit disturbance model is written, the system parameters specified in the Simplex algorithm are part of prior knowledge that is given to the control strategy and that the controller does not have to identify based on experiments.

- **The PBM [67], AMOCS [68] and AMOCS-MA [120]** approaches use implicit modeling requiring a very limited effort from the software engineer. The engineer should only specify the actuators and sensor, and possibly some weights that are unrelated to the model itself, but specify the properties of controller and how to reach the goals. Despite the lack of modeling needs from the software engineer, these approaches still build a representation of the software in the form of equations in their model-building phase. The synthesized model is then used to create a controller.

- **Advances in control theory have recently unveiled a new set of methods, denoted model-free control [71, 86, 36]**. Model-free control synthesis does not build a model of the system to be controlled but only uses data to optimize a control strategy. To date, model-free control has not been applied to software, and could open possibilities for performance improvement and to tackle the complexity of software systems in an automated way.
Apart from using the same type of model, all the automated approaches discussed in this chapter synthesize centralized control solutions deployed on a single software product. Such approaches are not suitable for systems where communication between components is limited or very costly. A recent work on architecture-based adaptation [177] introduced a number of patterns for designing decentralized adaptation solutions, where controllers make independent decisions, but have some kind of interaction. The automated control solutions may definitely benefit from this and similar efforts, as they provide means to adapt an entirely new class of software systems.

8.5 Conclusions

Throughout the recent years, the automatically generated control-theoretical solutions have made a huge progress. Starting from addressing a single adaptation requirement, these solutions can now handle multiple goals of different types, deal with addition or removal of system requirements on-the-fly or even adapt based on the predicted software evolutions. In this Chapter, we listed the key research steps that led to such progress and highlighted the main approaches representing each of the steps. Surely, the automated approaches have limitations. For example, they use simple models that are not always accurate and they are less effective in specific scenarios than controllers finely tuned for those scenarios. However, the main advantage of automated control comes from these limitations: simple models in combination with a generally applicable controller allow to build a control-based self-adaptive system without involvement of a control expert.

As for the future of automated control-based solutions, the research efforts can be aimed in two directions. First, as the scope of applicability and practical effectiveness of existing solutions is often unclear, these solutions should be tested in the industrial settings. Second, the researchers could use more state-of-the-art practices, such as model-free control or decentralized adaptation.
Chapter 9

SimCA*: Handling Uncertainty in Self-Adaptive Software Systems

In this Chapter, we present a control-theoretical approach to realize self-adaptive software called SimCA* (Simplex Control Adaptation Star) that satisfies STO-reqs in the presence of different types of uncertainties and provides adaptation guarantees.

SimCA* answers the research question QII-2 “How to tackle different types of uncertainty with an automated CBSA solution that deals with multiple requirements and provides adaptation guarantees?” by:

• Composing adaptation actions from multiple instances of SimCA (Chapter 6) into a global adaptation strategy, which allows to handle uncertainty in software component interactions.
• Monitoring changes in system requirements (including addition or removal of requirements at runtime) and updating the adaptation logic accordingly. For example, when a requirement is added, a new controller and a new Simplex equation are added to SimCA*. This allows SimCA* to deal with requirement uncertainty.
• Tracking changes of system parameters and updating the adaptation logic when needed. For example, if a network node starts over consuming energy, the equations of SimCA* are updated and the node gets less data packets.
• An extensive evaluation of SimCA* on an Internet of Things network, where system components are interrelated and communication between components is subject to disturbances of very high magnitude.

This chapter presents our article submitted to a journal [157]. Personal contribution: Stepan Shevtsov contributed for 75% to the developed the approach called SimCA*, he did 100% of the technical implementation (program code, testing, etc.), performed 100% of data collection, 75% of the analysis, and 60% of writing the manuscript.
SimCA*: A Control-Theoretic Approach to Handle Uncertainty in Self-Adaptive Systems with Guarantees

Abstract

Self-adaptation provides a principled way to deal with software systems’ uncertainty during operation. Examples of such uncertainties are disturbances in the environment, variations in sensor readings, and changes in user requirements. As more systems with strict goals require self-adaptation, the need for formal guarantees in self-adaptive systems is becoming a high-priority concern. Designing self-adaptive software using principles from control theory has been identified as one of the approaches to provide guarantees. In general, self-adaptation covers a wide range of approaches to maintain system requirements under uncertainty, ranging from dynamic adaptation of system parameters to runtime architectural reconfiguration. Existing control-theoretic approaches have mainly focused on handling requirements in the form of setpoint values or as quantities to be optimized. Furthermore, existing research primarily focuses on handling uncertainty in the execution environment. This paper presents SimCA* that provides two contributions to the state-of-the-art in control-theoretic adaptation: (i) it supports requirements that keep a value above and below a required threshold, in addition to setpoint and optimization requirements; and (ii) it deals with uncertainty in system parameters, component interactions, system requirements, in addition to uncertainty in the environment. SimCA* provides guarantees for the three types of requirements of the system that is subject to different types of uncertainties. We evaluate SimCA* for two systems with strict requirements from different domains: an Unmanned Underwater Vehicle system used for oceanic surveillance and an Internet of Things application for monitoring a geographical area. The test results confirm that SimCA* can satisfy the three types of requirements in the presence of different types of uncertainty.

9.1 Introduction

Modern software applications need, more than ever, being able to deal with change [49, 169]. The need for continuous availability of software applications requires developers to consider change as part of the development process. Software is expected to deal seamlessly with different types of uncertainty during operation. Examples of these uncertainties include disturbances in the environment such as noise, changes of values of system parameters such as varying accuracy of sensor readings, uncertainty in software component interactions, and changes in user requirements. Often, these uncertainties are difficult to predict
at design time, requiring software to be deployed with incomplete knowledge and handle changing conditions during operation [160, 171]. Consequently, software engineers are investigating new techniques to handle uncertainty at runtime without incurring penalties and downtime, which is commonly referred to as self-adaptation [98, 136, 43, 51, 169]. Many software systems today need to comply with strict requirements, providing guarantees for system properties such as ensuring a certain level of performance. It is then essential for these systems to be reliable and robust to disturbances [49, 163, 35, 175]. Control theory has been identified as one of the approaches to design adaptation solutions with formal guarantees [82, 25, 189, 50].

Self-adaptation in general covers a wide field of work, ranging from dynamic adaptation of system parameters for requirements satisfaction under uncertain operating conditions to runtime adaptation of components and architectural reconfiguration. Within this domain, a number of automated control-theoretic approaches for the adaptation of software have already been proposed [2]. These approaches mainly focus on dynamic adaptation of system parameters. In general, they are subject to two important limitations for practical applications. First, they satisfy only stakeholder requirements either in the form of setpoint values (S-reqs) or values to be optimized (O-reqs), e.g. see [67, 68, 154]. A typical example of a setpoint requirement for a Web server application is to keep the response time of service invocations at a required level. An optimization requirement for such setting is to reduce the overall operation cost. However, software systems today often need to address a third type of requirement: a threshold requirement that keeps a value above/below a threshold (T-reqs). A threshold requirement for the Web server example is to keep the failure rate of service invocations below a required threshold. In fact, a typical software adaptation problem would be to simultaneously satisfy a combination of S-reqs, T-reqs and O-reqs, which we refer as STO-reqs.

The second limitation of existing approaches is their support to deal with different types of uncertainty. Mahdavi et al defines uncertainty as “the circumstances when a software system behaviour deviates from the expected one due to various runtime dynamics and events that are difficult to predict at design time” [124]. The most common type of uncertainty is uncertainty in the environment in the form of disturbances. An example in the context of a Web server application is other software applications running on the same server that affect the server performance. Although most control-theoretic approaches can handle environment disturbances, other types of uncertainties are often not considered [143]. One type of such uncertainty is uncertainty in system parameters, where values of certain parameters of the software can fluctuate during operation. An example in the context of the Web server application is a change in the expected response time. To the best of our knowledge, the control-theoretic approach introduced in [67] is the only automated approach that deals with uncertainty in system parameters. Other automated approaches rely on fixed values for system parameters that are not updated at runtime. However, handling uncertainty in system parameters is important in practice as it ensures that accurate adaptation decisions are made (the details on uncertainty sources and types are given in Section 9.2).
Another type uncertainty that existing control-theoretic approaches do not support is *uncertainty in component interactions*, i.e. existing approaches are typically applied to systems where adaptation of one software component does not directly affect the adaptation of other components. E.g., in the Web server application, the choice of a particular service provider for one service will not influence the choice of the service provider for another service [154]. However, in many types of systems, especially in distributed settings, adaptations of the software components have interdependencies. Solving the adaption problem with one global adaptation strategy — i.e., selecting settings for all components simultaneously — may then become too complex or even infeasible. E.g., determining the settings of devices of an Internet of Things multi-hop network that send data over different paths needs to take into account the incoming traffic. Uncertainties about the load generated by each device makes it very hard to solve the adaptation problem with a global adaptation strategy. Hence, the adaptation problem needs to be partitioned and the solutions need to be composed to determine the overall adaptation configuration.

Finally, the existing approaches are limited in handling *uncertainties in requirements*. While most approaches provide basic support for changing the values of requirements, see e.g., [120, 154], they do not allow for the addition or removal of system requirements at runtime. Changing requirements is important in practice, e.g., to deal with drastic changes in the environment or the system itself that may require a change from one set of requirements to another.

In this paper, we apply control theory to deal with a typical adaptation problem for systems with strict goals: (i) to deal with multiple STO-reqs, (ii) to handle uncertainty in system parameters, component interactions, requirements and the environment, and (iii) to provide formal guarantees that the system complies with the requirements while operating under different types of uncertainties.

To address the formulated problem, we devised SimCA* (Simplex Control Adaptation1), an automated control-based approach for self-adaptive software systems that satisfy multiple STO-reqs. 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 operate on these models, and combines controller outputs using the simplex method to adapt the system. To deal with the different types of uncertainty, SimCA* has dedicated components that monitor changes in the system or its environment and adjust the adaptation logic accordingly. Hence, our work contributes to automated control-theoretic adaptation and is aligned with the scope of state-of-the-art research in this area [2, 157]. Dealing with adaptation at the level of the architecture of a system (changes of software interfaces, addition of new software modules, etc.) is out of scope of this work.

We conduct a formal analysis of controller properties of SimCA* to provide guarantees for controller stability, rejection of disturbances of certain magnitude, among others. This analysis is based on an equation-based model of the software system and leverages on guarantees provided by basic SimCA. The formal analy-

---

1The “*” symbol refers to the ability of the approach to handle different types of requirements and uncertainties.
sis is complemented with an empirical evaluation that demonstrates that SimCA* achieves the required quality goals. This evaluation is conducted on two cases from different domains: an Unmanned Underwater Vehicle (UUV) system that performs surveillance missions of a maritime environment and an Internet of Things (IoT) system used for monitoring of a geographical area. Both systems must self-adapt to guarantee the satisfaction of STO-reqs at runtime, while dealing with different types of uncertainties. The UUV case can be solved using a direct global adaptation strategy, while the IoT case requires the composition of local solutions to generate a global adaptation strategy. In order to evaluate the effectiveness of the approach, we also compare SimCA* with a state of the art architecture-based adaptation approach [88].

The SimCA* approach presented in this paper contributes the following:

1. It preserves the benefits of our initial work on basic SimCA [153, 1] that deals with multiple requirements and handles uncertainty in the environment in form of disturbances.
2. It handles uncertainty in system parameters by tracking changes of system parameters and updating the adaptation logic when needed.
3. It handles uncertainty in software component interactions by composing adaptation actions from multiple instances of basic SimCA into a global adaptation strategy.
4. It deals with requirements uncertainty by monitoring changes in system requirements and updating SimCA* accordingly.
5. In addition, we provide formal guarantees for controller properties of SimCA* based on an equation-based model of the software system, and complement that with and extensive evaluation of SimCA* on a complex case in the domain of Internet of Things, where components are interrelated and their communication is subject to disturbances of high magnitude.

The remainder of the paper is structured as follows. Section 9.3 positions SimCA* in the state-of-the-art automated control-theoretical approaches for self-adaptive software systems. Section 9.4 elaborates on the adaptation problem we address and illustrates it with a scenario. In Section 9.5 we provide a general overview of SimCA*. Section 9.6 summarizes basic SimCA, a building block of SimCA*. Section 9.8 describes how SimCA* handles uncertainty in component interactions. Section 9.9 explains runtime activation/deactivation/adjustment of requirements with SimCA*. Section 9.7 describes how SimCA* deals with uncertainty in system parameters. The formal guarantees provided by SimCA* are evaluated in Section 9.10. In Section 9.11, SimCA* is empirically evaluated with two cases. Finally, we draw conclusions and outline directions for future research in Section 9.12.

9.2 Focus of Study

This section describes the focus of our work in detail. We position our work in the field of self-adaptive systems in and in the field of control theory.
Self-adaptation and Uncertainty

The seminal article by Kephart and Chess [98] is one of the pioneering efforts that introduced the concept of self-adaptation in order to deal with the ever-growing complexity of the management of software systems. Since then, numerous researchers and engineers have studied and developed self-adaptive software systems using different techniques for a wide variety of application domains. Our work lays within the so-called “sixth wave of evolution of self-adaptation” [169]. In this wave, the self-adaptive systems are designed based on principles from control theory, aiming to provide guarantees on the behavior of the system that is subject to different types of uncertainty. According to [169], software systems have to deal with uncertainty coming from four main sources, see Table 9.1:

In this work, we deal with the following uncertainties (highlighted in the Table):

- Disturbances coming from the execution environment, such as noise and signal interference.
- Uncertainty in system parameters, where software parameters change at runtime due different internal or external factors.
- Uncertainty in component interactions. Here, uncertainty in component interactions means that an adaptation action performed by one system component may affect adaptations performed by other components.
- Requirement changes. In our case, we consider only anticipated uncertainty in requirements, meaning that the system goals that represent quality requirements (response time, failure rate) can be activated, deactivated, or their values may be adjusted at runtime based on conditions that are defined before deployment, but that can only be resolved during operation.

Human-related uncertainty (like human-in-the-loop, or multiple ownership of software elements) is out of scope of this work.

Control Theory

Control theory is a subfield of mathematics that provides tools and techniques to design and analyze self-adaptive systems. In particular, control theory provides basic ground on modeling a system and synthesizing a control strategy that adapts a system in order to achieve specific goals. The use of control-theoretical design to manage adaptation in computing systems has been researched for a couple of decades. Pioneering research in this direction is documented in [82].

Figure 9.1 shows the typical scheme of a self-adaptive software system designed based on principles from control theory. It employs the block diagram notation from control theory and depicts a feedback control loop. This feedback loop provides the basis for the core SimCA module described in Section 9.6. From left to right, the Setpoint represents the goal that the adaptation needs to achieve – i.e., a target value for a non-functional requirement such as a specific value for energy that can be consumed or a threshold for packet loss that can be tolerated. Based on the value of the desired goal and the corresponding Measured Output the Error is computed as: Setpoint – Measured Output (the -1 block indicates that
Table 9.1: Types of uncertainty (based on [169]). Uncertainties handled by SimCA* are marked in gray.

<table>
<thead>
<tr>
<th>Source</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment</td>
<td>Disturbances</td>
<td>Noise, signal interference and other types of disturbances might affect the system unpredictably.</td>
</tr>
<tr>
<td></td>
<td>Context change</td>
<td>The software execution context might change and evolve during operation.</td>
</tr>
<tr>
<td></td>
<td>Data sources</td>
<td>Usage of data from different sources might lead to uncertain conditions during operation.</td>
</tr>
<tr>
<td>Software</td>
<td>Parameters</td>
<td>Values of software parameters can change during operation.</td>
</tr>
<tr>
<td></td>
<td>Models</td>
<td>Uncertainty coming from the use of abstractions in modeling, inaccurate model representation of the system, model learning based on incomplete data, etc.</td>
</tr>
<tr>
<td></td>
<td>Component interactions</td>
<td>Decisions made by one software component might affect the work of other components unpredictably.</td>
</tr>
<tr>
<td></td>
<td>Decentralization</td>
<td>A system component may have very limited knowledge about the system state and states of other components.</td>
</tr>
<tr>
<td></td>
<td>Architecture</td>
<td>Some software parameters or components might be added or removed at runtime.</td>
</tr>
<tr>
<td>Requirements</td>
<td>Requirement elicitation</td>
<td>Formulating the system requirements based on stakeholder needs is not always straightforward, leading to uncertainty.</td>
</tr>
<tr>
<td></td>
<td>Requirement change (adjust/add/remove)</td>
<td>The stakeholder requirement values might change during operation according to new circumstances. The set of requirements might change as well, for example to adapt to a critical failure of one of the system components.</td>
</tr>
<tr>
<td>Human-related</td>
<td>Human-in-the-loop</td>
<td>Human factor is a known source of uncertainty. Decisions made by humans might be unexpected.</td>
</tr>
<tr>
<td></td>
<td>Ownership</td>
<td>The code of some system components may be owned by third parties and protected by copyrights, which might result in an uncertain behaviour at runtime.</td>
</tr>
</tbody>
</table>

the value of the measured output will be subtracted from the setpoint value). The Feedback Controller uses the error together with a model of the system to compute the Control Signal. The system model used by the controller describes the dynamics of the software system. In discrete time, this model is typically represented by a set of difference equations. In the line of research on automated control-theoretic adaptation [2, 157], where the work presented in this paper fits, the system model
is usually automatically identified during a learning phase. We elaborate on this in the next section. The computed control signal adapts the *Software System* such that the output gets as close as possible to the *Setpoint*.

![Figure 9.1: Basic diagram of a feedback control scheme.](image)

Note that the feedback control scheme may have a different structure depending on the problem at hand. In SimCA, we use multiple feedback controllers working in parallel that are connected to a Simplex block in a hierarchical structure. But generally speaking, most of the control strategies are developed to counteract the effect of *Disturbances* on the system. In case of software systems, these disturbances come from different sources of uncertainty that were discussed in Section 9.2.

**Guarantees**

The use of controllers in SimCA* provides a number of guarantees (see Figure 9.2):

- **Stability**: the ability of an adaptation mechanism to converge to S- or C-goals ($s_i/c_i$). Stability relates to most software qualities that are subject of adaptation. For example, lack of stability for an energy consumption goal means that the system may consume energy unpredictably;

- **Absence of overshoot**: the measured quality property does not exceed the goal $s_i/c_i$ before reaching its stable area. A non-zero overshoot leads to a penalty on the respective software quality. For example, an overshoot of a vehicle speed goal may lead to going above the speed limit and even breaking the vehicle;

- **Zero steady-state error**: the measured quality property does not oscillate around goal $s_i/c_i$ during steady state. Like stability, steady-state error is related to most software qualities that are subject of adaptation. For example, a non-zero error for an energy consumption goal means the system will constantly switch between underconsuming and overconsuming energy;

- **Tuneable settling time**: the time it takes to bring a measured quality property close to its goal $s_i/c_i$. Settling time influences most of the software qualities that are subject of adaptation as well. For example, having a low settling time for a battery consumption goal means spending less battery charge in a transient state;

- **Tuneable robustness**: the amount of perturbation in the environment that the system can withstand, while remaining in stable state. Robustness directly influences system reliability.
The formal analysis of these guarantees for SimCA* is provided in Section 9.10. These guarantees are based on an equation-based model of the software system and leverages on guarantees provided by basic SimCA. Section 9.11 complements the formal analysis with empirical data that confirms that the quality goals of the two systems that we use in the evaluation are achieved.

![Figure 9.2: Properties guaranteed by the controllers in SimCA*](image)

### 9.3 State of the art overview

There is a body of research available that applies principles from control theory to adapt software systems, for a recent survey see [2]. However, as shown in this survey, most of the proposed approaches tend to solve specific problems within a certain domains. Over the past years, researchers have investigated approaches that (semi-)automatically build a controller solution, aiming to create a reusable approach to build self-adaptive software that satisfies different stakeholder requirements with guarantees [157]. The work presented in this paper contributes to this line of research. We highlight representative work on automated approaches and position our work in this landscape.

One of the first contributions to create an automated control-theoretic approach for self-adaptation is the so called Push-Button Methodology (PBM) [67]. The main aim of PBM was to automate the design of a control theoretical adaptive system. PBM automatically creates a linear model of software and a controller that adapts the software to meet a non-functional requirement specified by the stakeholders. The main advantage of PBM is the assurance of a broad range of control-theoretical guarantees. The main limitation of plain PBM is that it supports only a single adaptation goal.

Follow-up research efforts studied and created automated solutions that satisfy multiple adaptation goals, e.g., to achieve a specific service response time and minimize the amount of service failures at the same time. In [68], Filieri et al. proposed an 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 one O-req. AMOCS maps the available actuators with the adaptation goals and creates a chain of controllers that use PBM to achieve these goals. As a result, the goals are prioritized based on their position in the chain. In other words, the second controller in the chain provides guarantees only if the first controller satisfied its goal. While this approach can
handle multiple goals, AMOCS may produce sub-optimal adaptation decisions, since it does not use all the available actuators for all the goals simultaneously.

In [154] we introduced basic SimCA, an approach that combines controllers with the simplex optimization algorithm in order to satisfy multiple S-reqs, while being optimal according to a single O-req. The controllers of SimCA are responsible for handling disturbances, while simplex solves the multi-objective optimization problem. This approach guarantees optimality of the solution and provides control-theoretical guarantees (stability, settling time, etc.) at the same time. In [1] we added support for T-reqs to basic SimCA and provided an initial, though ad-hoc, approach to support changing adaptation goals. As we explained in the introduction, SimCA provides a building block for SimCA* that we present in this paper, but contrary to SimCA*, basic SimCA does not support different types of uncertainties that are crucial for practical applications, including uncertainty in software component interactions, in system requirements, and in system parameters.

Recently, researchers investigated the use of Model Predictive Control (MPC) in control-theoretic software adaptation. In this direction, a semi-automated approach – Control-based Requirements-oriented Adaptation (CobRA) framework [10] – has been presented and has been followed by a fully automated alternative – Automated Multi-objective Control of Software with Multiple Actuators (AMOCS-MA) [120]. In these approaches, the controller acts based on the current feedback from the software, but uses the model of its own behavior to predict the evolution of the software system\(^1\). The use of MPC allows to achieve both optimality and most of the control-theoretical guarantees (e.g., stability, minimizing settling time), but requires a higher computation power. In case it is not possible to provide this computation power, a sub-optimal solution can be computed in a very limited amount of time, making the approach flexible also with respect to the characteristics of different problems. The main drawback of automated MPC is that the robustness guarantees are limited, i.e. the approach is sensitive to frequent disturbances and model inaccuracies.

In summary, existing approaches cannot deal with all of the following:

1. Address 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).
2. Handle requirements uncertainty (activation and deactivation) during system operation, which limits the applicability to practical software systems that are subject to continuous change.
3. Deal with uncertainty in the system parameters. In other words, if some system parameters change slightly during operation, the adaptation strategies will not update the system model and still work based on values received during the learning phase. This may lead to a less accurate or even incorrect solutions to the adaptation problem.

\(^1\)The same principle has been exploited also in non-control-theoretical solutions with satisfactory results [130, 131].
4. Handle uncertainty in component interactions. This is crucial for large scale and distributed systems, where applying a single adaptation strategy to adapt all the system components may be problematic or even impossible.

SimCA* on the other hand can satisfy STO-reqs in the presence of different types of uncertainty. To summarize the state of the art, we gathered the key properties of the main automated control-theoretical adaptation approaches presented in Table 9.2.

**Table 9.2: Automated Control-Theoretical Software Adaptation Approaches**

<table>
<thead>
<tr>
<th>Approach</th>
<th>Uncertainty Guarantees</th>
<th>Guarantees</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Requirement</td>
<td>Actuator</td>
</tr>
<tr>
<td>PBM</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>AMOCS</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>basic SimCA</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>AMOCS-MA</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SimCA*</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**9.4 Problem Definition**

Based on the analysis of the state of the art, we identified the following research problem:

_to guarantee the satisfaction of STO-reqs in the presence of uncertainty in system parameters, component interactions, requirements and environment._

Compared to state of the art approaches, four key challenges must be addressed in order to deal with the formulated problem. First, the solution must incorporate mechanisms to guarantee the satisfaction of STO-reqs. This is not trivial, in particular for T-reqs as these are not a typical type of requirement supported in control theory. Second, the solution requires a mechanism to integrate local adaptation decisions to handle interactions between software components. Third, the system should include a mechanism that monitors the relevant system parameters and adjusts the corresponding values used by the adaptation logic. Finally, the solution needs a mechanism to update the adaptation logic on the fly to address anticipated requirements changes, i.e., adjusting/activation/deactivation of requirements. SimCA* described in Sections 9.6-9.10 addresses these challenges.

**Problem example: DeltaIoT network**

We now describe a DeltaIoT system [164] that we use to illustrate the adaptation problem we aim to solve; this system is also used as a basis for one of the cases.

---

2 The characterization _part._ refers to partial support of the according feature.
for the evaluation of SimCA* in Section 9.11. The DeltaIoT system (DeltaIoT in short) is a distributed Internet of Things (IoT) application for monitoring a geographical area.

IoT is a rapidly evolving technology with applications for example in smart homes, smart grids, industry 4.0, and more general in smart cities. However, the implementation of high-quality IoT applications is challenging because:

- The capabilities of IoT devices are limited as they are typically small, cheap, and battery powered. On the other hand, these devices are expected to provide reliable communication without battery replacement for long periods. Designing a reliable IoT communication network that efficiently uses the available energy is a particularly important challenge, as communication is the primary energy consumer in IoT [6].

- Determining the optimal system configuration of an IoT network is challenging as the system is subject to various types of uncertainties at runtime. These uncertainties include interferences in the communication network, sudden changes in traffic load, mote malfunctioning, among others. Current practice to deal with uncertainties based on over-provisioning combined with manual tuning are not very efficient and expensive [164].

Hence, IoT is becoming an emerging and interesting domain for applying self-adaptation in general and control-theoretical adaptation in particular.

DeltaIoT consists of a set of tiny embedded computers (motes) that are placed in different buildings of Campus Arenberg at KU Leuven, Belgium, see Figure 10.1. Each mote is a system component equipped with a sensor for monitoring some property of the environment (e.g., movement or temperature). The motes can interact via the communication links between them. In particular, the motes communicate the sensor data via a LoRa-based multi-hop network to a gateway as final destination. The monitoring data is analyzed by an IoT application deployed at a server directly connected with the gateway that takes action if needed, e.g., by warning an operator.

DeltaIoT uses a time-synchronized communication protocol that divides the communication in cycles. Each cycle consists of a number of communication slots; in each slot one mote can communicate a number of packets with one other mote. The ordering of slots is organized such that data produced by the leaf motes of the network is sent first to their parents, these parent motes can then send these packets plus the locally produced messages to their parents, and so on until all packets in the network reach the gateway. For example, mote5 in Figure 10.1 first receives packets from mote10 and then sends packets to mote9 that then sends its data to the gateway.

The motes generate different numbers of packets per cycle depending on the type of sensor they use and the conditions in the environment. For example, a temperature sensor may take samples at a constant pace, while a movement sensor

---

1 https://www.lora-alliance.org/technology

2 In principle, motes that are out of each others communication range may be allocated slots in parallel.
may be very active during the day, but inactive in the evening. We refer to the property that expresses the probability that a mote generates packets during a cycle as its activation probability. The activation probability is expressed in %, where 100% means 10 data packets are generated per cycle, 0 means no packets are generated, while 40% means 4 packets are generated.

Each transmission of packets in the network consumes a certain amount of energy of the two motes involved (for sending and listening respectively). Transmitting a packet may fail, which is denoted as packet loss. The packet loss depends on the Signal-to-Noise ratio (SNR) of a wireless link during the communication. SNR represents the ratio between the level of the signal produced by the sending mote and the level of the interference and noise that comes from the environment. The packet loss can be reduced by increasing the SNR of the transmission over the link. However, this will require a higher power setting and thus more energy of the sending mote. Each link of the wireless network is characterized by a table of values where each available power setting is paired with the resulting basic SNR value of that link, plus an additional disturbance interval that expresses uncertainty in the environment, see Table 9.4 in Appendix for concrete values of DeltaIoT. The disturbance is defined as a value that is randomly selected from the disturbance interval and added to the basic SNR.

Finally, each mote has a limited queue for storing packets (incoming packets from its children plus its own generated packets). Consequently, sending too many packets to the same mote may cause a queue overflow and the loss of packets, which is denoted as queue loss.

As IoT networks are required to work reliably for a long period of time without replacing the battery, the main goal in DeltaIoT is to minimize the energy consumption of the motes, while ensuring a high packet delivery, i.e., keep the packet loss below a required threshold. These requirements can be achieved by tuning the power settings of the individual motes and/or by adjusting the paths along which
packets are transmitted. For example, assume that mote12 sends 50% of the packets to its parents (mote7 and mote3), each with a power settings of 3. If the link between mote12 and mote7 suffers from interference, then either the power setting of the communication with mote7 can be increased, e.g., to 5, or alternatively the distribution of messages sent to the parents may be changed temporally, e.g., 20% for mote7 and 80% for mote3.

In this study, the concrete requirements for DeltaIoT are as follows:

- **R1**: The average packet loss of the network should not exceed 5% over a period of 12 hours;
- **R2**: Subject to **R1**, the energy consumed by the motes should be minimized.

**R1** is a T-req, **R2** is an O-req. To illustrate the difference between T- and S-reqs, if the requirement would be to reach exactly 5% packet loss over a period of 12 hours, **R1** would become an S-req.

We refer to the scenario where DeltaIoT must satisfy requirements **R1** and **R2** as a *normal operation* mode, denoted by \( M_{no} = \{ R1, R2 \} \). We also consider a more challenging scenario, where the average packet loss should not exceed 2% over a period of 12 hours; denoted as **R1***. During busy hours, the traffic in the network may be very high, so packets may get lost because of full queues of some of the motes. Under such conditions, DeltaIoT may switch to *busy operation* mode, denoted by \( M_{bo} \) and defined as \( M_{bo} = \{ R1*, R2, R3 \} \). Requirement **R3** (T-req), that needs to be activated during operation, is defined as:

- **R3**: The average queue loss should be lower than 5% of packets sent over a period of 12 hours;

It is important to note that an adaptation solution for DeltaIoT needs to deal with the three types of uncertainties we consider in this research: uncertainties in system parameters (mote activation probabilities and fluctuating SNR of links), in component interactions (adaptation of packet distributions over a link affects how the packets should be distributed over all the following links of the same route), and in requirements (**R3** needs to be activated on the fly).

In summary, DeltaIoT is a system that is expected to meet strict requirements of stakeholders and deal with different types of uncertainty. This creates the need for self-adaption with guarantees.

### 9.5 Overview of Simplex Control Adaptation* – SimCA*

Figure 9.4 gives a high-level overview of SimCA*. The adaptation logic of SimCA* consists of four interrelated components. SimCA* takes as input the STO-reqs and operates on the Software System that is subject of adaptation to realize the requirements.

The *Multi-SimCA* component consists of a set of basic SimCA modules \((\text{SimCA}_{1..m})\) that deal with multiple STO-reqs. A basic SimCA module runs on
the fly experiments on the software system in an automated fashion, builds a set of linear models of the software system at runtime, creates a set of tunable controllers that operate on these models, and combines controller outputs using the simplex method to adapt the system. The basic SimCA module is discussed in details in Section 9.6. Note that, in the current realization, SimCA* adapts systems where STO-reqs can be directly assigned to basic SimCA modules. Additional work may be required to adjust SimCA* for more complicated interactions between system requirements and components.

SimCA* includes three components that deal with different types of uncertainty. The Actuation Signal Composer handles uncertainty in software component interactions by composing the adaptation actions generated by multiple SimCA modules into a global adaptation strategy. The Goal Updater deals with requirements uncertainty by monitoring anticipated changes in system requirements, and updating the running adaptation logic of basic SimCA modules accordingly. For example, when a new requirement is activated, a controller is added to the adaptation logic. The Parameter Updater deals with uncertainty in system parameters by tracking changes of those parameters and updating the values of according parameters of basic SimCA modules. The three components that deal with uncertainty are discussed in details in Sections 9.8-9.7.

Assumptions and scope of applicability. SimCA* targets a family of software systems that work under a number of assumptions. While these assumptions put restrictions on the target application domains, they hold for a large family of modern software systems. 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.).

- Has multiple possibly conflicting requirements 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 or disappear during operation.

• Does not have to deal with uncertainty related to humans-in-the-loop or multiple ownership of software elements.

9.6 Basic SimCA and Dealing with Disturbances

In this section we provide an overview of basic SimCA, that provides a core module of SimCA*. We start with a short introduction of basic SimCA and how it works in different phases. Then we elaborate on the different phases.

Introduction to Basic SimCA

Basic SimCA automatically builds a controller solution to satisfy a set of STO-requirements in the presence of disturbances (environment uncertainty). The approach comes with a set of formal guarantees.

SimCA requires: (i) a set of tunable parameters (actuators) that can be used to adapt the running system to address the requirements, and (ii) a set of adaptation sensors to measure the effect of the adaptation on the system. To apply SimCA the STO-reqs needs to be transformed into quantifiable goals (STO-goals). For example a requirement to keep the average response time $t$ at 3 ms is transformed to an S-goal $t = 3$ ms, or a requirement to maximize service task frequency $T_f$, while using not more than 1 MJ of energy $E$ will be transformed to O-goal $\text{max}[T_f]$ and T-goal $E \leq 1$ MJ.

SimCA works in four phases that are performed during system operation, see Figure 9.5.

• In the Identification phase, SimCA runs online experiments using sampled values of S- and T-goals to synthesize equation-based models of the software system.

• 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.

• In the Goal Transformation phase, the T-goals are transformed into controller goals (C-goals) using simplex. C-goals represent either the lowest possible value that satisfies all other goals (keep value below a threshold), or the highest possible value (keep value above a threshold).
• In the Operation phase, the controllers carry out control for the S- and C-goals; the controller outputs are combined with the O-goals using simplex to drive the system towards its goals.

Basic SimCA provides a core module for SimCA*. It allows solving a local adaptation problem of one software component, i.e. to adapt a component such that it satisfies a set of STO-requirements, while being robust to uncertainty in the environment. Basic SimCA comes with a set of formal guarantees that are inherited by SimCA*. We zoom in one these aspects in the following sections.

Phases of Basic SimCA

Phase I. Identification. In this 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 [154]. 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 between measurements during identification can be chosen by the system engineer, influencing the model quality [154]. 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 be factored into the model construction, they will be dealt with by using feedback from the running system.

Phase II. Controller Synthesis. In this 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 [154].

In SimCA, we use the following controller:

---

1 For problems that can be solved with a single global adaptation strategy, SimCA* requires only one basic SimCA module.
\[ u_i(k) = u_i(k-1) + \frac{1-p_i}{\alpha_i} \cdot e_i(k-1) \quad (C_i) \]

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 run-time; (2) a critical update mechanism, which allows reacting to unexpected critical changes in the system by triggering re-Identification [154].

**Phase III. Goal Transformation.** This phase transforms all T-goals into Controller goals (C-goals), see Figure 9.6. 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 the lowest possible value below the threshold, while satisfying all other goals. Different to an S-goal whose value is constant (except when the corresponding system requirement – S-req – changes), the value of a C-goal is updated after almost any change in the system, including parameter updates, adjustment, activation or deactivation of any requirement, etc. In other words, a value of C-goal that is optimal in current conditions will not be optimal if the system changes. Under these new conditions the C-goal needs to be recalculated using simplex as discussed below.\(^2\)

The transformation of goals is required as SimCA controllers cannot work with T-goals by design, while the use of Simplex without controllers will lead to the loss of formal guarantees provided by SimCA. 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.

\[ \text{Figure 9.6: Goal Transformation phase of SimCA.} \]

\(^2\)Intuitively, an S-goal enables a stakeholder to express a specific value for a requirement, e.g., the service response time should be 6 sec. A T-goal on the other hand enables a stakeholder to express a threshold for a requirement, e.g., the service response time should be below 6 sec. During the transformation of a T-goal to a C-goal, simplex will find an optimal value for the C-goal that complies with the threshold requirement, given the actual conditions. E.g., under certain conditions, simplex may find a C-goal = 1 sec for the best service response time, which is six times better as a solution with an S-goal of 6 sec, while under other conditions simplex may find a C-goal = 2 sec, which is still three times better as a solution with an S-goal.
Generally, simplex allows finding an optimal solution to a linear problem written in the following standard form

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

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 \[47\].

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 \( P(k) \) of the system (i.e., relevant parameters of system components that can be measured\(^3\)). 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 (1) and for a proof of its optimality we refer to the linear programming literature \[47, 46, 144\].

Knowing the vector \( x \), each T-goal is transformed into C-goal \( c_i \) as follows: \( c_i = P_i(k) \times x \). As simplex takes into account all STO-goals of the system and it was formally proven to find the optimal solution to systems of equations such as (1), it guarantees that the calculated value of C-goal is the most optimal for the current system conditions, hence satisfying the corresponding T-goal in the most optimal manner. 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 this 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 goals, see Figure 9.7. As simplex is dealing with the O-goals, only C-goals obtained during Goal Transformation and original S-goals are used in the Operation Phase.

![Figure 9.7: Operation phase of SimCA (illustrated for one S- and C-goal).](image)

In particular, SimCA collects all control signals \( u_i(k) \) and the system parameters \( P(k) \) and passes these to simplex. Similarly to the Goal Transformation phase, SimCA solves the system of equations (1) to find a solution (actuation signal \( u_{sx} \))

\(^3\)E.g., in DeltaIoT \( P(k) \) are the average SNR value of different routes.
that drives the system towards an output that satisfies all STO-goals. However, the system of equations (1) has now a slightly different structure. First, each equation, except the last one, now represents an S-goal or a C-goal to be satisfied. Second, 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. Third, the constants \( b \) in (1) are replaced by control signals \( u_i(k) \) obtained from \( C_i \), providing all the advantages of the controllers.

### 9.7 Handling Uncertainty in System Parameters

In order to deal with uncertainty in system parameters, SimCA* is equipped with a Parameter Updater component. The Parameter Updater measures and records a set of parameters \( P \) during the Operation phase. The Parameter Updater analyses this data using a change-point detection algorithm. When a change point is detected, the corresponding system parameter in \( P \) is updated to a new value, see Figure 9.8. For example, in DeltaIoT, SimCA* records the SNR values of all links and the activation probabilities of all motes. When the change-point detection algorithm detects a significant change in these parameters, the corresponding input parameters for simplex are updated.

![Figure 9.8: Dealing with changing system parameters in SimCA*.](image)

The change-point detection problem has extensively been studied in data mining research [183, 97]. The core of this problem is to detect a point on a time-series where the data changes significantly. Change-point detection has been used to deal with a variety of problems, including intrusion detection in security systems, fault detection in software products, big data analysis, among many others.

There are many methods available for change-point detection [7]. In SimCA* we use a variant of the so-called likelihood ratio method, because it is simple, tunable and effective enough to solve the problem of handing uncertainty in system parameters. This method selects a certain point in a time series and uses statistical analysis on the data of a particular interval in the past (before the point) and an interval in the present (after the point). The selected point is considered a change point if the distribution of the data in the two intervals is significantly different [97], i.e. when the ratio between the averages of the values in the intervals (i.e., the likelihood ratio) is above a certain threshold.
An advantage of the likelihood ratio method is that it is an unsupervised method, so it works in different scenarios without the need for prior learning. The method has two tunable parameters: the length of the interval in the past/present used for detection (also known as time window length); and a threshold that is used when calculating the likelihood ratio (known as decision threshold). As proven by [178], these two parameters provide trade-offs between time delay of change-point detection, the probability of false alarm, and the probability of correct detection of a jump of a certain magnitude. Based on experimental results, in SimCA* we use an interval that is 10 times bigger than the adaptation period and a threshold that is 5% of the difference between the maximum and the minimum values of the subject concern. For example, if the route SNR changes between -2 and 2, the SNR threshold will be \[(2 - (-2)) \times 0.05 = 0.2.\]

### 9.8 Handling Uncertainty in Component Interactions

In order to handle uncertainty in software component interactions, i.e., uncertainty that arises from an adaptation action performed by one system component affecting adaptations performed by other components, SimCA* applies a modular approach. Namely, every system component that may affect the work of other components is equipped with an own instance of basic SimCA that calculates a local actuation signal \(u_{sx}\). Then, SimCA* is equipped with an Actuation Signal Composer that calculates a resulting global actuation signal \(u_{gl}\) based on all local actuation signals and the type of interactions between software components, see Figure 9.9. As such, the changes in component interactions do not influence the internal structure of basic SimCA modules nor the composition of controllers. SimCA* is not able to automatically handle drastic changes in component interactions, i.e. the network structure changes. In that case, the system engineer will need to adjust the Actuation Signal Composer to compose actuation signals from basic SimCA modules accordingly.

SimCA* offers a generic approach to handle uncertainty in component interactions using an Actuation Signal Composer. As component interactions are application-specific, it is the task of the system engineer to instantiate the Actuation Signal Composer for a concrete application case at hand. When designing a concrete Actuation Signal Composer, a set of rules needs to be followed. First, the Actuation Signal Composer should automatically adjust the global actuation signal \(u_{gl}\) when any of the local actuation signals \(u_{sx}\) change. For example, if at runtime one component doubles the consumption of a certain resource that is shared among other components, the Actuation Signal Composer should adjust the global adaptation strategy accordingly. Second, in order to preserve the formal guarantees provided by basic SimCA, the Actuation Signal Composer should produce a setting that satisfies all local actuation signals without changing them. For example, when composing two local actuation signals, one can apply each of the signals for a certain period of time. Third, the Actuation Signal Composer should be able to handle conflicts between local actuation signals. For example, when two components want to use a specific resource simultaneously, the Actu-
We illustrate how the Actuation Signal Composer handles uncertainty in component interactions for the DeltaIoT example (Section 9.4). In DeltaIoT, the distribution of packets over different routes is calculated for motes 7, 10 and 12, see Figure 10.1. This results in local actuation signals $u_{sx(7)}$, $u_{sx(10)}$ and $u_{sx(12)}$. The variables of these signals represent the packet distribution over possible routes. E.g., $u_{sx(12)}$ consists of four variables, representing the packet distribution over routes 12-7-2-4-1, 12-7-16-1, 12-7-3-1 and 12-3-1, see Figure 10.1. Changing this local actuation signal to calculate distribution of packets for parent links only (i.e., 12-7 and 12-3) is not possible, because such distribution will not account for the parameters of the following links in a route. In other words, sending all packets via link 12-3 because of its low packet loss, will violate the network packet loss requirement due to high packet loss at link 3-1.

Obviously, local actuation signals $u_{sx(12)}$ and $u_{sx(7)}$ are conflicting, as they both set packet distribution probabilities over links 7-2, 7-16 and 7-3. So the Actuation Signal Composer will first calculate the amount of packets distributed by mote12 $pd_{12}$ and mote7 $pd_{7}$ based on activation probabilities of those motes and their children. I.e., $pd_{12}$ is a sum of activation probabilities of motes 12, 14 and 15, while $pd_{7}$ is a sum of activation probabilities of motes 7, 13 and 11. The resulting global actuation signal will be set as follows: $u_{gl(7)} = pd_{7}/(pd_{7} + pd_{12}) \times u_{sx(7)} + pd_{12}/(pd_{7} + pd_{12}) \times u_{sx(12)}$. In summary, the Actuation Signal Composer allows multiple basic SimCA modules to work in parallel, each module associated with a mote that has multiple parents. Each SimCA distributes only the packets that arrive at and are generated by the mote associated with the module.

1 As all other motes have only one parent link, all packets are sent over these links.

2 The calculation performed by Actuation Signal Composer is actually more complex, because it takes into account a number of specific factors, such as multiple routes of the same basic SimCA that include the same link. As our focus is not on the details of the algorithm, we refer the interested reader to the SimCA* project website [155]
9.9 Dealing with Requirements Uncertainty in SimCA*

This section describes how SimCA* adapts the system when requirements are changed (activated, deactivated or adjusted values) during operation. It is important to note that SimCA* supports anticipated uncertainty regarding requirements, i.e., the approach allows to activate, deactivate, and adjust requirements on the fly based on conditions that are defined before deployment but that can only be resolved during operation. Examples are: a user decides to activate an extra requirement or the system faces a sudden change that leads to a change of goals. To that end, we extended the workflow of the basic SimCA modules (see Figure 9.5) with an additional Goal Update Phase, see Figure 9.10.

![Figure 9.10: Dealing with requirement changes in SimCA*](image)

Any change of requirements during system operation triggers the Goal Update Phase. In this phase, the running adaptation logic is updated according to the change in requirements. For example, when a new requirement is activated, a new controller and a new simplex equation is added to the adaptation logic of the basic SimCA modules. Depending on the type of requirement change, the system makes a transition from Goal Update to either Identification or Goal Transformation. In the remainder of this section, we explain in detail the activation and deactivation of requirements, and the change of requirement types. As adaptation logic in response to changing requirements is the same for all basic SimCA modules, for clarity we explain Goal Update with a single basic SimCA module.

### Requirement Activation

To deal with the activation of a new requirement, the Goal Updater component of SimCA* triggers a sequence of actions as shown in Figure 9.11. First, the Requirement Monitor subcomponent tracks changes in the system requirements. Depending on the type of changed requirement, it then triggers the S-/T-goal Activator or the O-goal Activator subcomponent. The Goal Activator first transforms the new requirement into a quantifiable goal (see Section 9.6) and reads the relevant
parameters $\mathcal{P}$ related to that goal\(^1\). In case the O-goal Activator is triggered, it inserts $\mathcal{P}$ into the objective function $c^T$ of simplex, performs a Goal Transformation (Section 9.6) and proceeds to standard Operation. In case the S-/T-goal Activator is triggered, it adds an equation for the new S-/T-goal to the system (1) to be solved by simplex; this equation has the same structure as the equations that represent the other S-/T-goals, see Section 9.6. After that, the S-/T-goal Activator performs an Identification for the new 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, S-/T-goal Activator triggers Controller Synthesis in order to build a controller for the new goal, followed by a Goal Transformation, after which the system returns to standard Operation.

**Requirement Deactivation and Changing Requirement Types**

For a requirement deactivation or a change of requirement type, the Goal Updater identifies the required change and depending on that triggers a sequence of actions as shown in Figure 9.12.

Similarly to requirement activation, the Requirement Monitor subcomponent tracks the system to identify the need for a change of the requirements. Depending on the type of changed requirement it triggers the S-/T-goal Deactivator or the O-goal Deactivator.

\(^1\)For example, if the queue loss requirement of DeltaIoT ($R_3$) is activated, the Goal Activator reads the activation probabilities of all motes. We discuss details on controlling queue loss in DeltaIoT in Section 9.11.
In case the O-goal Deactivator is triggered, it removes the variables of that goal from the objective function $c^T$ of simplex. In case the S-/T-goal Deactivator is triggered, it removes the controller and the equation from simplex that corresponds to the deactivated goal. Finally, both deactivators trigger 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 during operation. To that end, SimCA* performs the following: (i) if an S-req is changed to a T-req (or vice versa), the corresponding equality is changed to inequality in the system of equations (1), followed by a Goal Transformation; (ii) if an S-/T-req is changed to an O-req, the parameters $\mathcal{P}$ 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).

Changing requirement types at runtime allows the system to continue working in a number of additional scenarios. For example, if at one point during operation a stakeholder would like to minimize packet loss in DeltaIoT, while consuming not more than a certain amount of energy, the system will not require a complete restart, but just change the types of both requirement.

### 9.10 Formal Evaluation of Guarantees

SimCA* inherits a broad set of guarantees provided by basic SimCA. The guarantees provided by the controllers include (see Figure 9.13):

- **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**: the time it takes to bring a measured quality property close to its goal $s_i/c_i$;
- **Tuneable robustness**: the amount of perturbation in the environment that the system can withstand, while remaining in stable state.

Since the T-goals, that are expressed as inequalities, are transformed to equalities (C-goals) during Goal Transformation (see Section 9.6), simplex that works with these equalities during Operation, does not introduce additional system dynamics. Instead, it applies a straight-forward translation of the control signals to an actuation signal. Furthermore, as the Actuation Signal Composer (see Figure 9.4) composes the local actuation signals of the basic SimCA modules to a global actuation signal that is applied to the Software System without changing them, the
guarantees provided by basic SimCA hold for SimCA*. Hence, 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 the control system used in SimCA* [154]:

\[
O_i(k) = s_i \cdot (1 - p_i^k)
\]  \hspace{1cm} (2)

During steady-state time goes to infinity, \( k \to \infty \), and since \( p \in (0, 1) \) we get \( p_i^k \to 0 \) in this case. The steady-state error \( \Delta e \) is then:

\[
O_i(k \to \infty) = s_i \cdot (1 - p_i^k) = s_i; \hspace{0.5cm} \Delta e = s_i - O_i = 0
\]

In [154] we derive the relation between settling time \( \bar{K} \), robustness \( \Delta(d) \) and pole \( p_i \):

\[
\bar{K} = \frac{\ln \Delta s_i}{\ln |p_i|} \hspace{0.5cm} 0 < \Delta(d) < \frac{2}{1 - p_i}
\]  \hspace{1cm} (3)

In other words, a lower value for \( p_i \) leads to weaker disturbance rejection but faster response to change. Note that in the equations (2) and (3) \( s_i \) can be replaced with \( c_i \) without any effect on the guarantees as C-goals represent particular values (setpoints) to be achieved by the system, similar as S-goals.

Regarding the guarantees when requirements or the system parameters are changed, we assume that those changes will not lead to an unfeasible solution. Under this assumption the guarantees will hold, because changing the number of controllers or simplex equations will not alter the structure of the adaptation logic.

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 [154]).

Simplex provides the following guarantees:

- **Optimality**: the 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 9.6 [47, 46].
- **Scalability**: a small amount of extra time and effort is required to solve problems of growing scale. For practical problems, simplex usually finds a solution in just a few iterations [45]. This also ensures that the overhead is low for requirement changes as only one extra simplex iteration is required.
9.11 Experimental Evaluation

• Detection of an infeasible solution: the 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 $-\infty$). An 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 system model; if the system is not able to identify a sufficiently good model (for example, when the model cannot sufficiently represent the system non-linearities) then the controller will not be able to achieve its goals and guarantees. To ensure that the model reflects the dynamics of the real system, SimCA* performs identification 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 a requirement should not lead to an unfeasible solution (see discussion above). Third, the guarantees are provided after the controllers are built, i.e., control-theoretical guarantees apply only during the Operation phase. Fourth, 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 architecture is changed.

9.11 Experimental Evaluation

We empirically evaluate SimCA* with two cases. First, this Section describes the experimental setting of the UUV case with different STO-reqs. In this case, adaptation is realized by SimCA* equipped with a single basic SimCA module. Then we apply SimCA* to this case and experimentally demonstrates the guarantees and quality trade-offs provided by SimCA* under different operating conditions. After that, this Section describes the setup of the second case with DeltaIoT. In this case, adaptation is realized by SimCA* equipped with a multiple basic SimCA modules. We show the results of SimCA* applied to this case and compare SimCA* with a state of the art architecture-based adaptation approach. We also perform experiments with DeltaIoT when requirements change and a new requirement is activated at runtime. Finally, this Section 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 SimCA* project website [155].

Experimental Setting: UUV System

First, we show the core functionality of SimCA* (Section 9.5) on a case of the UUV system [151]. UUVs are increasingly used for a wide range of tasks. 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., these vehicles are expensive equipment that should work accurately and productively, and they should not impact the ocean area or get lost during missions.

The UUV in our study is used to carry out a surveillance and data gathering mission, e.g., to monitor the pollution of a maritime area. The system is implemented in a Java simulation environment. The UUV 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, consuming a certain amount of energy, see Table 9.3. The sensor data in this table is subject to a randomly distributed disturbance up to 10%. A scan is performed every second. The sensors being used during missions are selected by a single software component deployed on the UUV.

**Table 9.3: 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>

The UUV system has the following requirements:

- *R1*: A segment of surface scanned by the UUV when traveling over a distance of \( S \geq 100 \text{ km} \) should be examined in \( t = 10 \text{ hours} \); \(^1\)
- *R2*: To perform the mission, a given amount of energy \( E = 5.4 \text{ MJ} \) is available;
- *R3*: Subject to *R1* and *R2*, the accuracy of measurements should be maximized.

In other words, the UUV should examine as much surface as possible using all the available energy, while ensuring maximum accuracy. *R1* is a T-req, *R2* is a S-req, while *R3* is an O-req.

To realize the requirements *R1* – *R3*, SimCA* is deployed and operates on top of the software component that controls UUV sensors and turns them on and off during a mission. We assume that only one sensor is active at a time, but, SimCA* uses a combination of sensors during each adaptation period. As there is only one key component involved in the adaptation, the UUV system in our

---

\(^1\)When an UUV moves and takes scans, the sensors scan a particular area beneath the vehicle with a fixed width. Hence we can keep the requirement simple by expressing it in terms of traveled distance.
study does not have uncertainty in component interactions and therefore uses only one basic SimCA module. However, the UUV has to deal with uncertainty in the environment (sensor failures, noise in data communication channel), uncertainty in system parameters (runtime changes of UUV sensor parameters presented in Table 9.3) and uncertainty in requirements (adjustment of requirements $R_1 - R_3$ during operation).

SimCA* performs adaptations every 100 surface measurements of the UUV system, i.e., the time instance $k$ is incremented by 1 every 100 measurements. The application collects the UUV data to build performance graphs, which are used to evaluate SimCA* (see the experimental evaluation). The $x$-axis of the graphs are time instants $k$. The $y$-axis shows the average values of the measured feature per 100 surface measurements of the system. The implementation details of the UUV case are available at the SimCA* project website.
Adaptation with STO-reqs, Guarantees and Trade-offs

Figure 9.14 shows the adaptation results of SimCA* applied to the UUV system configured according to the experimental setting described above. The controller pole $p$ is set to 0.6. Adaptation starts with the Identification phase that is clearly visible for $k$ between 0 and 21. The Control Synthesis phase, immediately followed by the Goal Transformation phase, starts after the relationships between control signals $u_i(k)$ and system outputs $O_i(k)$ are identified ($k = 22$). For comparison, the “Scanning Speed” plot contains an additional line (see “Threshold” in Figure 9.14) representing 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. Since $R_1$ is a T-req, i.e., the UUV must scan $S \geq 100$ km, SimCA* looks for a combination of sensors that allows to scan more surface without losing accuracy or spending extra energy during the Goal Transformation phase (see Section 9.6). SimCA* uses simplex to find an optimal solution, which in this scenario is scanning 3.2 meters of surface per second. As such, the scanning speed goal is transformed from T-goal $V \geq 2.7$ to C-goal $V = 3.2 \text{ m/s}$.

After the goal is updated, the Operation phase starts (from $k = 22$ onwards). The two upper plots in Figure 9.14 show that the system is stable during Operation, i.e., the measured energy consumption and scanning speed follow their goals. To demonstrate how SimCA* deals with requirement uncertainty, we adjust the available energy two times: 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 the new conditions. Note that reducing the available energy at $k = 100$ increases the scanned distance (speed changes from 3.2 to 3.55 m/s) but decreases measurement accuracy (from 92.7% to 89.4%).

Figure 9.14 shows how SimCA* reacts to uncertainty in system parameters and uncertainty in the environment (a UUV sensor failure in this case). At $k = 220$, the energy consumed by sensor $S_1$ increases from 170 to 190 J/s. To deal with this overconsumption, a portion of the time allocated to $S_1$ is given to sensor $S_3$ that consumes less energy, 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 $89.94 \pm 0.04\%$.

To experimentally verify the guarantees and quality trade-offs provided by SimCA*, we perform the same experiment using controllers with pole $p = 0.9$, see Figure 9.15. 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 $89.94 \pm 0.04\%$.

The different graphs for both pole settings show that the UUV system is stable, has a zero steady-state error, and converges to the goals without overshooting. The results confirm that the system requirements are satisfied.
As described in Section 9.10, adaptation with SimCA* is influenced by the values of the pole $p$. A smaller pole leads to a shorter settling time. In particular, the settling time $\bar{K}$ of controller $C_i$ depends on the pole $p_i$ and a constant $\Delta s_i$ chosen by the system engineer: $\bar{K} = \ln \frac{\Delta s_i}{\ln p_i}$. According to [82, 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 number of adaptation steps required to obtain a change of amplitude 1 in the measured value of a goal determining the setting time. For example, the settling time can be observed at $k = 100$ on the “Scanning Speed” plot of Figures 9.14 and 9.15 where the speed is required to change from 3.14 to 3.58 m/s (change of amplitude 0.44). Then, $\bar{K}_{0.6} = 7.66 \times 0.44 = 3.4$ steps and $\bar{K}_{0.9} = 37.3 \times 0.44 = 16.4$ steps. These values explain why the measured scanning speed makes almost a vertical jump at $k = 100$ in Figure 9.14, while in Figure 9.15 it takes 17 adaptation steps to converge to a target value.
By comparing the experiment outcomes obtained from 50 runs, we can conclude that a smaller pole leads both to a larger 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 transition phase.

However, note that a lower value of the pole of the controllers is not always a better option as it leads to a reduced rejection of disturbances. Due to a small amplitude in noise in the test scenario, both controllers successfully rejected disturbances. This may not be the case under different operating conditions, e.g., when a UUV would be subject to underwater streams, pressure, etc. Besides, a smaller pole makes the adaptation mechanism react faster not only to goal changes but also to disturbances. This property can be observed, for example, by comparing the usage curve of sensor $S_2$. In Figure 9.15 it is smoother and has a much lower spike at $k = 220$ than in Figure 9.15. In this case a slower reaction may be a benefit as it allows to switch less frequently between different sensor combinations.

**Experimental Setting: DeltaIoT**

We use the DeltaIoT network described in Section 9.4 as a case to apply SimCA* in a distributed setting and to evaluate its features to deal with uncertainty in requirements, component interactions, and system parameters. In this paper we use a Java simulation environment of the real IoT network that is deployed at the KU Leuven Campus. The main motivation for this choice is time constraints. While SimCA* produces solutions within orders of seconds (scalability provided by Simplex), the actual DeltaIoT system needs ~8 minutes for one communication cycle. In other words, using the real setup for the evaluations of SimCA* described in this paper would be impractical as each of the experiments would require a run for a period between 6 and 16 days in real time. Moreover, the actual DeltaIoT system and the simulator are fully compatible offering an identical monitor and actuator interface. Hence, the simulation environment allows to model and study the behavior of the IoT network in an efficient way. Note that the code size of SimCA* (~35Mb including the simulator) may become a problem for the tiny IoT devices, this problem can be solved easily by adding a simple controlling board equipped with the basic SimCA module to every node that is involved in component interactions. However, solving this technical problem is out of our scope of this paper.

All the parameter values of links and motes, including the disturbances of links and the activation probabilities of motes are available in Appendix. The data in these tables are based on data collected from field experiments. As explained in Section 9.4, the basic SNR of a link depends on the chosen power setting of the source mote, with 0 being minimum power and 15 maximum power. The SNR disturbance for each link is specified as an interval of values. At a particular point in time, the actual SNR of a link consists of the basic SNR of the link plus a randomly selected value from the SNR disturbance interval. For example, the link between mote2 and mote4 (link 2-4) with power setting 0 has a basis SNR of $7.0 - 5.0 = 2.0$ and a disturbance interval $[-5..5]$. Consequently, the actual SNR
of that link may range from \(7.0 - 5.0 = 2.0\) to \(7.0 + 5.0 = 12.0\). Note that for the link between mote10 and mote17 we use a disturbance profile as the disturbance measured for this link has a specific form. The default activation probability for the motes is set to 100\%, emulating a highly loaded network. The Appendix lists the motes (5, 7, 11, and 12) for which we used different activation probabilities and associated disturbance intervals.

In our experiments, adaptation is performed every 10 network cycles, i.e., the time instance \(k\) is incremented by 1 after every 10 cycles. Each cycle takes around 8 minutes in real time, but only a small fraction of this time is required in simulation. In each adaptation step the application calculates the average measured value of the \(i\)-th goal (e.g., packet loss) during the past 10 cycles. Then it calculates the error \(e_i\) as the difference between \(i\)-th setpoint (e.g., target packet loss) and the measured value of the \(i\)-th goal. The application also monitors the energy consumed for communicating packets, and the changes of system requirements.

The task of SimCA\(^*\) is to keep the packet loss of the network below a certain threshold over a period of 12 hours, while minimizing the network energy consumption. SimCA\(^*\) achieves this task by calculating the value of the global actuation signal, which represents the percentage of packets that should be sent via the different routes of the network (different routes correspond to different paths that can be selected to communicate packets from the leaf motes of the network to the gateway). The global actuation signal is based on local actuation signals \(u_{sx}\) coming from basic SimCA modules installed on every system component (mote) that affects the adaptation of other motes. In this scenario basic SimCA is installed on motes 7, 10 and 12; an example of calculating \(u_{gl}\) based on \(u_{sx}\) is given in Section 9.8. The system requirements are directly assigned to each of the basic SimCA modules of SimCA\(^*\). It means that, for example, to achieve 5\% packet loss on average, SimCA\(^*\) will not try to lose 10\% of packets at Route1 and 0\% at Route2; instead, both routes will be required to maintain a 5\% packet loss. During a period of 12 hours, SimCA\(^*\) performs 9 adaptations, so even if the network packet loss exceeds the threshold due to disturbances during one of the adaptation periods, the following adaptation action will adjust the routes to reduce the packet loss. Therefore, the average network packet loss over a period of 12 hours will not surpass the threshold.

The controller pole \(p_i\) is set to 0.9, which allows to reject disturbances of high magnitude; the choice of pole values is discussed in Section 9.11. The value of \(\delta\) is set at \((\max_i - \min_i) \times 0.1\).

In the following sections, we present the evaluation results. The simulator collects data from runs over periods of 5.5 days \((k = 100)\) in the first set of experiments and 16.5 days \((k = 300)\) in the second set of experiments. This data is used to build performance graphs, which are used to evaluate SimCA\(^*\). The \(x\)-axis of the graphs are time instants \(k\). The \(y\)-axis shows the average values of the measured properties per 10 network cycles. As the requirements \(R1\) (packet loss) and \(R3\) (queue loss) are defined in terms of averages over a period of 12 hours, the graphs also show these average values.
Adaptation of DeltaIoT with SimCA*

We first compare the results for packet loss and energy consumption of SimCA* and the reference approach (marked “Ref.” on Figure 9.16)\(^2\). In the reference approach the power of each mote in the network is set to maximum and all packets are forwarded via all available parent links. This conservative approach is commonly used in the industrial IoT networks [176]. The rationale of the reference approach is to give preference to high reliability (low packet loss) over shorter lifetime of the network (high energy consumption).

The left part of Figure 9.16 shows the results for normal operation mode, i.e., \(M_{n,o} = \{ R_1, R_2 \} \) (Section 9.4). The right part of the figure shows the results with a packet loss threshold set to 2\% (\(R_1^*\)), which is more challenging.

---

\(^2\)Packet loss (and queue loss) is expressed in percentages and energy consumption is expressed in Coulomb, or C in short.
As shown in Figure 9.16, the non-adaptive reference approach is able to keep the packet loss very low during normal operation mode (average packet loss 2.9% ± 0.5 over 100k), but at the cost of consuming a lot of energy (average 41.4C ± 0.3). SimCA* in this scenario achieves the packet loss requirement (average 4.2% ± 1.0). The results of SimCA* for packet loss comply with the requirement, but they are not as good as those of the reference approach. On the other hand, SimCA* outperforms the reference approach on energy consumption (average 16.6C ± 0.4 compared to 41.4C ± 0.3). The reference approach is not able to deal with the more challenging requirement of a packet loss threshold of 2% (average 2.9% ± 0.5 over 100k), while SimCA* is able to realize this requirement (average 1.3% ± 0.8). Besides this benefit, SimCA* also consumes less than half of the energy required by the reference approach (average 19.5C ± 0.6 for SimCA* compared to 41.4 ± 0.3 for the reference approach).

In conclusion, SimCA* is able to achieve the system requirements for the two scenarios, and the approach clearly outperforms the reference approach in terms of energy consumption.

**SimCA* vs Architecture-based Adaptation**

We compare now SimCA* with ActivFORMS, a representative state-of-the-art architecture-based adaptation approach [88] that adapts the system using a Monitor, Analyze, Plan, and Execute (MAPE) feedback loop [98]. With ActivFORMS, the system engineer first creates a model of the MAPE feedback loop. In ActivFORMS, the MAPE feedback loop is modeled using a network of timed automata by instantiating a set of model templates [48]. The feedback loop model is verified against a set of properties to ensure that the MAPE loop works correctly. The verified model is then directly deployed on top of a virtual machine that executes the automata models to realize adaptation. This approach allows avoiding extra coding and ensures functional correctness of the feedback loop with respect to a set of correctness properties.

ActivFORMS maintains and reasons over runtime models of different quality properties to make adaptation decisions. These models are verified using runtime statistical model checking in order to provide guarantees for the adaptation goals with a certain level of confidence. In short, the approach calculates all the possible adaptation options (system configurations), verifies the expected qualities of these options, and selects the best option that satisfies all the requirements. Given that ActivFORMS uses verification at runtime and the time to make adaptation decisions is constrained, the number of options that can be analyzed within the available time is bounded. For example, in DeltaIoT, there are 3 motes that have alternative routes to distribute their packets (from 0 to 100%). When a fine-grained resolution is used to distribute the packets over the routes (e.g., per 1%), the number of configurations that needs to be checked at runtime becomes very high. In practice, to get runtime results from ActivFORMS within the available time to compute adaptation options, we had to limit the resolution to distribute packets among links to 3 alternatives: 0%, 50% or 100%. I.e., in case of mote12, there will be only three adaptation options where link 12-7 is used to transfer 0%, 50%
or 100% of packets from mote12 (and link 12-3 is used to transfer rest of the packets). As a result, ActivFORMS has 108 adaptation options for the DeltaIoT setup that is used in the experiments.

To ensure that we used the same random disturbances for both approaches, we recorded the actual SNR values of all links and activation probabilities of all motes during each cycle when using SimCA*, and then used those recordings in the experiments with ActivFORMS.

Figure 9.17 presents the results of SimCA* and ActivFORMS. As in the previous experiments, the left part of Figure shows the adaptation results in normal operation mode $M_{no} = \{R1, R2\}$, while the right part shows the results with a packet loss threshold of 2% ($R1^*$).

The left part of the Figure shows that in normal operation mode ActivFORMS and SimCA* produce similar results for the packet loss (average over $K = 100$ is $4.2\% \pm 1.0$ for SimCA* versus $4.0\% \pm 1.1$ for ActivFORMS), and energy con-
sumption (average 17.5°C ± 0.8 for SimCA* versus 16.6% ± 0.4 for ActivFORMS). These outcomes show that for such experimental scenario, the distributions of packets over links of either 0% or 100% (as selected by ActivFORMS) produces sufficient results.

There is a difference in results between both approaches for the scenario with the more challenging requirement of a packet loss threshold of 2%. The right part of Figure 9.17 shows that both approaches achieve the requirements, but with slightly better results for SimCA* both for packet loss (average over $K = 100$ is 1.3% ± 0.8 for SimCA* versus 1.8% ± 0.6 for ActivFORMS) and energy consumption (average 19.5°C ± 0.6 for SimCA* versus 20.9°C ± 0.5 for ActivFORMS). The small benefits of SimCA* over ActivFORMS can be explained by the adaptation options that are available to both approaches. While SimCA (that can distribute packets with any resolution) distributes around 55% of packets to link 7-3 and 45% of packets to link 7-2, ActivFORMS is more restricted (it has to select between 0, 50, or 100%) and distributes 100% of the packets to link 7-2. On the other hand, both approaches agree that link 12-3 should be used over link 12-7.

In conclusion, the experimental results show that SimCA* and ActivFORMS achieve the requirements for both scenarios, but for more challenging requirements SimCA* is able to produce slightly better results. ActivFORMS is a representative example of architecture-based adaptation that uses runtime verification to select adaptation. The test results show that such approaches can be limited in terms of the adaptation options they can handle, which may lead to sub-optimal solutions. SimCA* on the other hand is significantly more efficient. By using simplex the approach is able to analyze very large spaces of adaptation options to produce optimal solutions.

Adding a New Requirement: Queue Loss

So far our evaluation discussed only the loss of packets due to interference of communication links. However, packets may also get lost due to an overflow of queues at the motes. In this section, we add a new requirement to the DeltaIoT system at runtime to deal with queue loss.

Figure 9.18 presents the adaptation results of SimCA* when the new requirement is added to the system. We start from a setting where the packet loss threshold is set to 2% ($R_1*$ and $R_2$). As the adaptation happens during busy hours when all motes are actively producing data, the test results show that around 11% of the packets are dropped due to overloaded queues. To prevent this loss of packets, DeltaIoT enters the busy operation mode $M_{bo} = \{R_1*, R_2, R_3\}$ at $k = 100$, where requirement $R_3$ (T-req) is activated. Recall that $R_3$ is defined as: the average queue loss should be lower than 5% of packets sent over a period of 12 hours.

To control the queue loss of the network, SimCA applies the following strategy. First, the activation probabilities of all motes are used to calculate the network link load, which represents the maximum number of packets that could be sent via each of the network links. Second, for each of the available routes, each basic SimCA calculates the route load by summing the load of links in that route. Finally, each
basic SimCA chooses the routes for sending data packets based on the route load: the lower the queue loss requirements, the lower route load is preferred.

The experiment starts with requirements $R_1^*$ and $R_2$ active. At $k = 100$, the new T-req $R_3$ is activated, see Figure 9.18. This triggers a new Identification phase from $k = 100$ to 110. During this phase SimCA* creates a model for the queue loss goal, adds a new controller to the system for the queue loss goal, and adds a new inequality to simplex. After Identification, the system returns to the Operation phase (from $k = 110$ onwards). The results show that even though the threshold for queue loss is set at 5%, SimCA* finds communication routes that satisfy the packet loss requirement (average $1.65\% \pm 0.7$ for $110 < k < 200$), while producing almost no queue loss (average $0.06\% \pm 0.2$).

To test the adjustment of a requirement, we changed the packet loss threshold from 2% ($R_1^*$) to 12% at $k = 200$. In response to this relaxation of the packet loss requirement, SimCA* changes the power settings of the motes resulting in a substantial reduction of energy consumption, see Figure 9.18 (the energy consumption decreases from around $19C$ to around $15C$ from $k = 200$ onwards). At the same time, the network routes are adjusted; more packets are sent over link 7-3 and less over link 7-16. This test scenario shows how SimCA* is able to support a trade-off between requirements; packet loss and network energy consumption in this case. Relaxing the packet loss requirement allows SimCA* to reduce the
energy consumption by increasing the traffic via a link that requires the mote to use a lower power setting (link 7-3), without violating the queue loss requirement.

**Adaptation to Changing system parameters**

To conclude, we experimentally evaluate the dynamic change of actuator settings. We start from a setting with requirements $R_1^*$ (packet loss threshold 2%) and $R_2$ (minimize energy consumption). Similarly to the experiment discussed in the previous section, we activate the queue loss requirement ($R_3$), this time at $k = 50$, see Figure 9.19.

As in the previous scenario, the results show that SimCA* is able to satisfy both threshold goals ($50 < k < 150$). However, at $k = 150$, we suddenly decrease the actual SNR of link 7-3 by a value of 20. The effect of this change of system parameter is that link 7-3 is no longer used for transmitting packets (from $k = 150$ onwards). In the last part of the experiment ($k = 230$) we introduce another change of the packet loss requirement, this time the threshold is set to 6%. As expected, SimCA* reacts to this change in requirement by adapting the packet distribution over alternative routes.

Notice that the drop in SNR of link 7-3 at $k = 150$ prevents SimCA* from using this link, which leads to zero queue loss. If we compare the usage of link 7-3 in this scenario and the scenario shown in Figure 9.18, one can conclude that the
queue loss of the entire DeltaIoT network is almost proportional to the usage of link 7-3. This is a result of a constant use of link 12-3 at 100% that saturates the queue of mote3.

The results of this experiment show that SimCA* is able to handle on the fly activations of new requirements and changing actuator settings. The approach dynamically adapts to system to deal with these uncertainties, while addressing the system requirements.

**Threats to Validity & Limitations**

SimCA* handles one class of adaptation problems (satisfying multiple STO-reqs with guarantees in the presence of different types of uncertainty), which apply to a significant number of software systems. At the same time, the approach should not be used on systems undergoing drastic changes in their behavior at runtime as continuous re-identification is very costly. Also, SimCA* in its current realization cannot deal with runtime changes of software architecture and software evolution.

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 or changing requirements that lead to unfeasible solutions (e.g., to satisfy R1, the system is forced to ignore R2). However, when requirements are interrelated (e.g., increase in R1 leads to decrease in R2), SimCA* will find a solution if it is feasible. We used standard controller guarantees and described their boundaries in Section 9.10. We also provided an initial mapping of controller guarantees to software quality properties in Section 9.2. However, additional research is required both to refine and extend this mapping and to understand the coverage of these guarantees.

We evaluated SimCA* in two domains, focusing on adaption for a typical set of stakeholder requirements (resource usage, performance, reliability). 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., actuator uncertainty 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 [59, 33, 31]. However, the deployment of SimCA* in a real-world setting is required to confirm the obtained results in practice.

**9.12 Conclusions**

In this paper we presented SimCA*, an approach that allows building self-adaptive software systems that satisfy multiple STO-reqs in the presence of different types of uncertainty. 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 solution that
adapt 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.

SimCA* was evaluated in a simulated distributed setting which confirmed the
ability of the approach to handle uncertainty in component interactions. This is an
initial step towards a completely distributed and decentralized control-based ap-
proach for self-adaptive software, which we plan to investigate in future research.
Also, in order to confirm the obtained result in practice, we are planning to ap-
ply SimCA* to a new version of the physical setup of DeltaIoT that is currently
deployed at the campus of KU Leuven.

**Appendix: DeltaIoT setup**

**Table 9.4: SNR values of DeltaIoT links**

<table>
<thead>
<tr>
<th>Link</th>
<th>Link SNR according to power setting (0-15)</th>
<th>Disturbance</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-4</td>
<td>7.0 7.6 7.8 7.6 7.6 7.3 7.3 7.8 7.8 7.6 7.3 7.9 6.9 7.3 7.9 8.0 [-5..5]</td>
<td></td>
</tr>
<tr>
<td>3-1</td>
<td>0.2 1.0 2.0 2.9 3.0 4.0 5.0 6.0 6.0 6.0 6.8 7.3 7.4 7.1 7.5 7.63 [-2..2]</td>
<td></td>
</tr>
<tr>
<td>4-1</td>
<td>-8.7 -7.8 -6.8 -5.8 -4.9 -3.8 -2.9 -1.9 -0.7 0.0 0.8 1.1 2.0 2.6 3.0 3.0 [-2..2]</td>
<td></td>
</tr>
<tr>
<td>5-9</td>
<td>-6.0 -4.9 -4.0 -3.1 -2.1 -1.1 -0.7 0.0 0.1 1.0 1.0 1.1 1.5 1.5 1.5 1.5 2.6 [-2..2]</td>
<td></td>
</tr>
<tr>
<td>6-4</td>
<td>0.2 1.0 2.0 2.9 3.0 4.0 5.0 6.0 6.0 6.0 6.8 7.3 7.4 7.1 7.5 7.6 [-2..2]</td>
<td></td>
</tr>
<tr>
<td>7-2</td>
<td>-3.0 -2.1 -1.0 0.0 0.1 1.0 1.4 2.0 2.9 3.0 4.0 4.0 4.6 5.0 5.0 5.0 [-5..5]</td>
<td></td>
</tr>
<tr>
<td>7-3</td>
<td>-7.9 -6.8 -5.9 -4.9 -4.1 -3.3 -2.4 -1.8 -0.9 -0.3 0.0 0.3 0.4 0.8 0.8 [-5..5]</td>
<td></td>
</tr>
<tr>
<td>7-16</td>
<td>-0.7 -0.4 -0.1 0.2 0.5 0.8 1.1 1.4 1.7 2.0 2.3 2.6 2.9 3.2 3.5 3.8 [-5..5]</td>
<td></td>
</tr>
<tr>
<td>10-5</td>
<td>-3.5 -3.0 -2.0 -1.0 0.0 0.0 1.0 2.0 3.0 3.0 4.0 5.0 5.0 5.0 5.0 [-5..5]</td>
<td></td>
</tr>
<tr>
<td>10-17</td>
<td>Custom SNR profile, see Figure 9.20</td>
<td></td>
</tr>
<tr>
<td>11-7</td>
<td>-4.0 -3.0 -2.0 -1.0 0.0 0.5 1.0 2.0 3.0 4.0 5.0 5.0 6.0 6.0 6.0 [-2..2]</td>
<td></td>
</tr>
<tr>
<td>12-3</td>
<td>6.0 6.1 6.2 6.3 6.4 6.5 6.6 6.7 6.8 6.9 7.0 7.1 7.2 7.3 7.4 7.5 [-2..2]</td>
<td></td>
</tr>
<tr>
<td>12-7</td>
<td>-13 -13 -12 -12 -12 -11 -11 -9.7 -8.9 -7.9 -6.7 -5.8 -4.9 -4.0 -3.1 -3.0 [-2..2]</td>
<td></td>
</tr>
<tr>
<td>13-11</td>
<td>-3.8 -2.8 -2.3 -1.3 -2.0 -1.0 0.0 0.0 0.0 3.3 3.7 4.0 4.0 4.3 4.3 4.7 [-5..5]</td>
<td></td>
</tr>
<tr>
<td>14-12</td>
<td>-6.6 -5.1 -4.2 -3.3 -2.6 -1.6 -1.0 -0.1 0.0 0.8 1.0 1.0 1.0 1.0 1.0 [-5..5]</td>
<td></td>
</tr>
<tr>
<td>15-12</td>
<td>-8.3 -7.3 -6.4 -5.6 -4.6 -3.8 -3.1 -2.3 -1.6 -0.9 -0.3 -0.5 -0.1 -0.1 0.3 0.4 [-2..2]</td>
<td></td>
</tr>
<tr>
<td>16-1</td>
<td>-0.2 0.1 0.4 0.7 1.0 1.3 1.7 1.9 2.3 2.6 2.9 3.2 3.5 3.9 4.2 4.5 [-5..5]</td>
<td></td>
</tr>
<tr>
<td>17-1</td>
<td>-3.8 -2.8 -2.3 -1.3 -2.0 -1.0 0.0 0.0 0.0 3.3 3.7 4.0 4.0 4.3 4.3 4.7 [-2..2]</td>
<td></td>
</tr>
</tbody>
</table>
Figure 9.20: Custom SNR profile of link 10-17

Table 9.5: DeltaIoT motes with non-default activation probabilities.

<table>
<thead>
<tr>
<th></th>
<th>mote5</th>
<th>mote7</th>
<th>mote11</th>
<th>mote12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activation probability, %</td>
<td>80</td>
<td>80</td>
<td>80</td>
<td>90</td>
</tr>
<tr>
<td>Disturbance, %</td>
<td>[-10..10]</td>
<td>[-20..20]</td>
<td>[-10..10]</td>
<td>[-5..5]</td>
</tr>
</tbody>
</table>
Research Summary and Road Ahead

The final part of this dissertation provides summary of the research results and directions for future work. It consists of Chapter 10 that discusses the research contributions and some of the open research possibilities.
Chapter 10

Conclusions

In this final Chapter we draw conclusions from the conducted research. In particular, the research contributions are presented in Section 10.1, while Section 10.2 outlines directions for future work.

10.1 Contributions

In this thesis, we studied how to realize self-adaptive software systems that satisfy multiple stakeholder requirements with guarantees in the presence of uncertainties. 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 [2]. The survey describes 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. This literature review serves as a foundation for all the following contributions.

- SimCA: a reusable control-based engineering approach that allows to build self-adaptive software systems that satisfy different types of requirements [154, 153, 1]. SimCA directly addresses RPI as it can also deal with disturbances and requirement changes at runtime. SimCA includes a formal model of software systems and an adaptation solution combining controllers with the simplex algorithm to handle multiple requirements. The formal evaluation of SimCA deals with the part of RPI regarding the guarantees. During the experimental validation we applied SimCA to systems with strict requirements to measure the achieved software qualities (performance, reliability, etc.) and to confirm that the guarantees hold in those systems.

- SimCA*: a reusable control-based engineering approach that builds upon SimCA and handles different types of uncertainty [156]. SimCA* addresses RPII by introducing new components that deal with uncertainty in software parameters, addition or removal of requirements at runtime and software component interactions. During the experimental validation, we applied SimCA* to the UUV system and the DeltaIoT network to confirm the approach functionality and obtained guarantees.
10.2 Future Work

Future efforts within the line of research presented in this dissertation could include two major goals. We discuss these goals in the following sections.

Ongoing Work and Short-term Goal

Our short-term goal is to apply experience gained from the conducted research in an Internet of Things setting. We are already working with our partners from KU Leuven, VersaSense company, University of York and Lund University on a framework to realize self-adaptive Internet of Things systems that address multiple stakeholder requirements and deal with different types of uncertainty. We plan to use the developed framework to realize a network deployed in KU Leuven. This network mimics an industrial IoT network setup and is called the Mesh network.

Mesh network is a low-power wireless multi-hop network that can autonomously achieve a long battery lifetime on IoT devices [126]. Despite its advantages, the in-build Mesh network stack does not take into account other system requirements (latency, frequency of measurements, etc.) and limits the adaptation possibilities only to the software application level, as the hardware and middleware levels of the communication stack are out of control of a software developer. I.e., we can change the allocation of bandwidth or frequency of tasks performed at a certain device, but we cannot decide how to route packets or schedule network communications.

In our case, Mesh network serves as a distributed Internet of Things (IoT) application for monitoring the working environment. This network consists of a set of tiny embedded computers (motes). The motes are placed in different rooms of the Computer Science department at KU Leuven, Belgium. Each mote is equipped with a sensor for monitoring some property of the environment. The motes have different characteristics depending on the type of equipped sensor. For example, sensors of type A consume more energy. The motes communicate the sensor data to a gateway $GW$ via links of the Mesh network. The monitored data is analyzed by an IoT application deployed at a server directly connected with the $GW$ that takes action if needed, e.g., by warning an operator. Figure 10.1 shows an overview of the part of deployed network with 12 motes. All links in the network are bi-directional and they are also constantly changed by the underlying network protocol, so the Figure just shows an example topology at a particular time instant.

![Figure 10.1: Mesh network deployment, 12 motes example.](image-url)
In order to make our network environmental friendly and to reduce the effort required to replace batteries, all motes of Mesh Network are equipped with a small solar panel connected to a capacitor, which allows to harvest energy during the day and then spend it in the evening and at night.

Each mote \( m \) has a number of configurable parameters that change its behavior. For example, one can allocate more bandwidth to \( m \) which will increase its energy consumption, but decrease latency of packets coming from \( m \). The Mesh network also has some configurable parameters that have an effect on the entire network. For example, when we increase the Number of parents \( NumPar \) parameter, each mote will get an additional parent (a link to another mote), leading to increased average energy consumption of the network, but decreasing the average network latency. The Mesh network has multiple requirements that can be satisfied by adapting parameters of motes and the network. For example, in order for the network to be functional for \( D \) days (requirement \( R1 \)), we need to tune parameters in such a way that none of the energy harvesting devices runs out of capacitor charge for at least \( D \) days.

The adaptation solution for Mesh network also needs to deal with the different types of uncertainties: environment uncertainty (changing weather conditions, interferences, noise in the data communication channels), software/structural uncertainty (changing links between motes, changing preferred routes to transfer data, motes joining and leaving the network), uncertainty in system goals due to changing requirement values.

In summary, Mesh network is a distributed system that is expected to meet strict requirements of stakeholders and deal with different types of uncertainty. This creates the need for self-adaption with guarantees. We plan to summarize our work on the Mesh network in a research publication and submit it to a journal in the first half of 2019. The results of this collaborative work have the potential to change the way of engineering self-adaptive IoT systems.

**Long-term Goal**

A long-term goal for future work would be development of a reusable control-theoretic solution that could solve the research problem \( RP \) in the entire family of distributed and heterogeneous systems. As our experiments with DeltaIoT showed, the manually adjusted and tuned SimCA* can be applied to some of those systems. Our ongoing work in the IoT domain also looks at handling multiple requirements in the presence of uncertainty in distributed systems. However, those are only first steps towards synthesizing a generally applicable solution for different types of distributed and heterogeneous systems. Creating such an approach remains a major open challenge that we could try to tackle in future.


