SimCA*: A Control-theoretic Approach to Handle Uncertainty in Self-adaptive Systems with Guarantees

STEPAN SHEVTSOV, Linnaeus University, Sweden, and KU Leuven, Belgium
DANNY WEYNS, KU Leuven, Belgium, and Linnaeus University, Sweden
MARTINA MAGGIO, Lund University, Sweden

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 article presents SimCA*, which 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.

CCS Concepts: • **Computing methodologies → Computational control theory**; • **Computer systems organization → Self-organizing autonomic computing**; • **Software and its engineering → Formal methods**; *Designing software*;

Additional Key Words and Phrases: Software, uncertainty, self-adaptation, control theory, SimCA*, IoT, UUV

ACM Reference format:
Stepan Shevtsov, Danny Weyns, and Martina Maggio. 2019. SimCA*: A Control-theoretic Approach to Handle Uncertainty in Self-adaptive Systems with Guarantees. *ACM Trans. Auton. Adapt. Syst.* 13, 4, Article 17 (July 2019), 34 pages.
https://doi.org/10.1145/3328730

Authors' addresses: S. Shevtsov, SE-35195, Universitetsplatsen 1, Linnaeus University, Växjö, Sweden; email: stepan.shevtsov@lnu.se; D. Weyns, SE-35195, Department of Computer Science and Media Technology, Linnaeus University, Växjö, Sweden; email: danny.weyns@kuleuven.be; M. Maggio, SE-221 00, Department of Automatic Control, Lund University, Sweden; email: martina@control.lth.se.

1556-4665/2019/07-ART17 $15.00
https://doi.org/10.1145/3328730

1 INTRODUCTION

More than ever, modern software applications need to be able to deal with change [14, 39]. 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 [37, 41]. 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 [9, 13, 24, 29, 39]. 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 [8, 14, 38, 40]. Control theory has been identified as one of the approaches to design adaptation solutions with formal guarantees [5, 15, 19, 45].

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 [33]. 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 References [17, 18, 34]. 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 define 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" [26]. 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 [30]. 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 Reference [17] 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 2.1).

Another type of 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 [34]. 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, existing control-theoretic approaches are limited in handling *uncertainties in requirements*. While most approaches provide basic support for changing the values of requirements (see, e.g., References [25, 34]), 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 article, 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 Adaptation[1]), 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 [33, 36]. 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 analysis 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

[1]The "*" symbol refers to the ability of the approach to handle different types of requirements and uncertainties.

to generate a global adaptation strategy. To evaluate the effectiveness of the approach, we also compare SimCA* with a state-of-the-art architecture-based adaptation approach [21].

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

(1) It preserves the benefits of our initial work on basic SimCA [34, 35], which deals with multiple requirements and handles uncertainty in the environment in the 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 an 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 article is structured as follows: Section 3 positions SimCA* in the state-of-the-art automated control-theoretical approaches for self-adaptive software systems. Section 4 elaborates on the adaptation problem we address and illustrates it with a scenario. In Section 5, we provide a general overview of SimCA*. Section 6 summarizes basic SimCA, a building block of SimCA*. Section 8 describes how SimCA* handles uncertainty in component interactions. Section 9 explains runtime activation/deactivation/adjustment of requirements with SimCA*. Section 7 describes how SimCA* deals with uncertainty in system parameters. The formal guarantees provided by SimCA* are evaluated in Section 10. In Section 11, SimCA* is empirically evaluated with two cases. We draw conclusions and outline directions for future research in Section 12.

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 Section 2.1 and in the field of control theory in Section 2.2.

2.1 Self-adaptation and Uncertainty

The seminal article by Kephart and Chess [24] 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" [39]. 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 Reference [39], software systems have to deal with uncertainty coming from four main sources (see Table 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 to different internal or external factors.

Table 1. Types of Uncertainty (Based on Reference [39])

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

Uncertainties handled by SimCA* are marked in gray.

- 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 (such as human-in-the-loop or multiple ownership of software elements) is out of scope of this work.

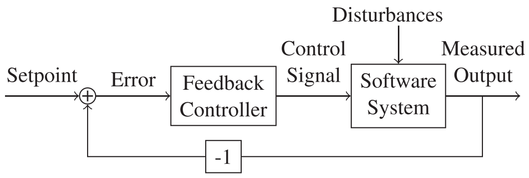


Fig. 1. Basic diagram of a feedback control scheme.

2.2 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 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 Reference [19].

Figure 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 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 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 [33, 36], where the work presented in this article 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.

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 2.1.

2.3 Guarantees

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

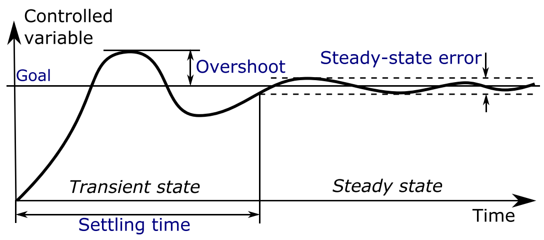


Fig. 2. Properties guaranteed by the controllers in SimCA*.

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;

- Tunable 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;
- Tunable 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 10. These guarantees are based on an equation-based model of the software system and leverages on guarantees provided by basic SimCA. Section 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.

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 Reference [33]). However, as shown in this survey, most of the proposed approaches tend to solve specific problems within certain domains. Over 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 [36]. The work presented in this article 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) [17]. 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 Reference [18], Filieri et al. proposed an approach for Automated Multi-Objective Control of Self-adaptive software (AMOCS). 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 Reference [34], we introduced basic SimCA, an approach that combines controllers with the simplex optimization algorithm 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 Reference [35], 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 article, 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 [4]—has been presented and has been followed by a fully automated alternative: Automated Multi-objective Control of Software with Multiple Actuators (AMOCS-MA) [25]. 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.[2] 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.
(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*, however, 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 2.[3]

[2]The same principle has been exploited also in non-control-theoretical solutions with satisfactory results [27, 28].
[3]The characterization *part.* refers to partial support of the according feature.

Table 2. Automated Control-Theoretical Software Adaptation Approaches

Approach	Adapt. Goals				Uncertainty			Guarantees				
	Types			Prio-	Requi-	Para-	Compo-	Stabi-	Settling	Over-	Robust-	Opti-
	S	T	O	ritiz.	rement	meter	nent Int.	lity	time	shoot	ness	mality
PBM	1					✓		✓	✓	✓	✓	
AMOCS	n		1	✓				✓	✓			
basic SimCA	n	n	1		part.			✓	✓	✓	✓	✓
AMOCS-MA	n		1	✓				✓	✓	✓	part.	✓
SimCA*	n	n	1		✓	✓	✓	✓	✓	✓	✓	✓

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.*

We explained the different types of uncertainties and the bounds of our work for each of them in Section 2. Compared to state-of-the-art approaches, four key challenges must be addressed 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 6–10 addresses these challenges.

Problem Example: DeltaIoT Network

We now describe a DeltaIoT system [20] 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 for the evaluation of SimCA* in Section 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. However, 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 [2].
- 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 expensive and not very efficient [20].

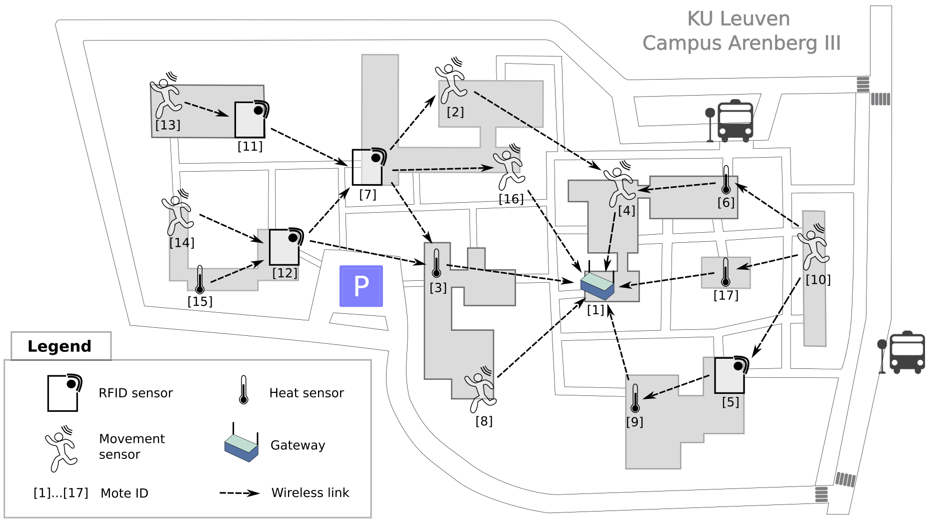


Fig. 3. DeltaIoT network topology.

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 3). 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.[4] 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.[5] 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 3 first receives packets from mote10 and then sends packets to mote9, which 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 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

[4]https://www.lora-alliance.org/technology.
[5]In principle, motes that are out of each other's communication range may be allocated slots in parallel.

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 4 in the 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 setting 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), which 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.

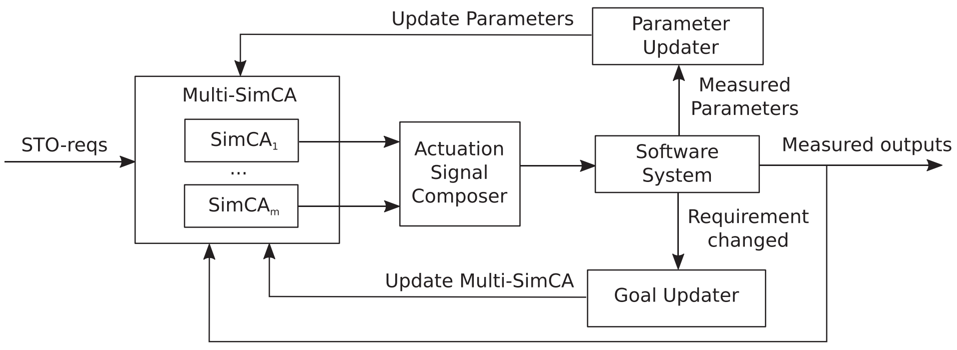


Fig. 4. General overview of SimCA*.

5 OVERVIEW OF SIMPLEX CONTROL ADAPTATION*—SIMCA*

Figure 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 (SimCA$_{[1..m]}$) that deal with multiple STO-reqs and handle disturbances. 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 detail in Section 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 *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 *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 three components that deal with uncertainty are discussed in detail in Sections 7 and 8.

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.

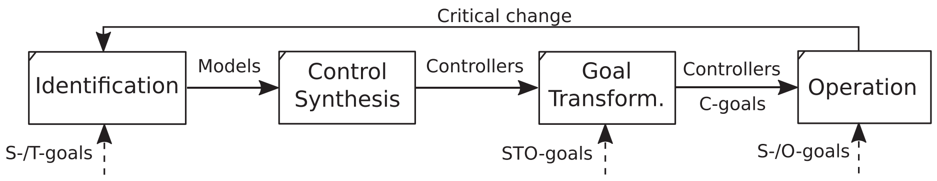


Fig. 5. Phases of SimCA.

- 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.

6 BASIC SIMCA AND DEALING WITH DISTURBANCES

In this section, we provide an overview of basic SimCA, which 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.

6.1 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 3ms is transformed to an S-goal $t = 3$ms, or a requirement to maximize service task frequency T_f, while using not more than 1MJ of energy E will be transformed to O-goal $max[T_f]$ and T-goal $E \leq 1$ MJ.

SimCA works in four phases that are performed during system operation (see Figure 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.[6] Basic SimCA comes with a set of formal guarantees that are inherited by SimCA*. We zoom in on these aspects in the following sections.

6.2 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 the form of control signals that effect software) and the measured system outputs [34]. Each model $\mathcal{M}_i$ 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 $\mathcal{M}_i$ 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:

$$O_i(k) = \alpha_i \cdot u_i(k-1). \tag{$\mathcal{M}_i$}$$

Coefficient α_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 [34]. Model $\mathcal{M}_i$ describes the system behavior ignoring small disturbances and sudden system changes. As small disturbances are difficult to predict at design time and to 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 [34].

In SimCA, we use the following controller:

$$u_i(k) = u_i(k-1) + \frac{1-p_i}{\alpha_i} \cdot e_i(k-1). \tag{C_i}$$

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 α_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.

In addition to disturbances, the controller C_i handles inaccuracies in the model $\mathcal{M}_i$. To that end, each controller incorporates: (1) a Kalman filter adapting the linear model at runtime; (2) a critical update mechanism, which allows reaction to unexpected critical changes in the system by triggering re-Identification [34].

Phase III. Goal Transformation. This phase transforms all Threshold goals (T-goals) into Controller goals (C-goals) (see Figure 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, and so on. In other words, a value of C-goal that is optimal in current conditions will not be optimal if the system

[6]For problems that can be solved with a single global adaptation strategy, SimCA* requires only one basic SimCA module.

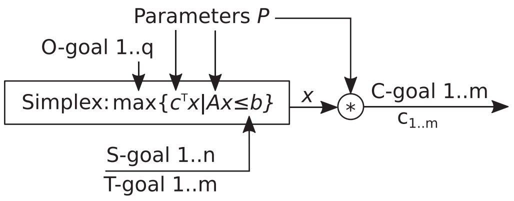


Fig. 6. Goal transformation phase of SimCA.

changes. Under these new conditions, the C-goal needs to be recalculated using simplex, as discussed below.[7]

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.

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

$$\max\{c^T x \mid Ax \leq b; x \geq 0\}, \tag{1}$$

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 [11].

In the Goal Transformation phase of SimCA, each equation except the last one represents an S-goal or T-goal to be satisfied. Equalities are used for S-goals, while inequalities are used for T-goals. The last equation ensures that the system selects a valid solution by constraining the values that can be taken by elements of the vector x, e.g., $x \geq 0$. The values of S-/T-goals to be achieved replace constants b, whereas matrix A and vector c^T are substituted with the monitored parameters $\mathcal{P}(k)$ of the system (i.e., relevant parameters of system components that can be measured[8]). 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 [11, 12, 31].

Knowing the vector x, each T-goal is transformed into C-goal c_i as follows: $c_i = \mathcal{P}_i(k) * x$. 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 Equation (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 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.

[7]Intuitively, an S-goal enables a stakeholder to express a specific value for a requirement, e.g., the service response time should be 6s. A T-goal, however, enables a stakeholder to express a threshold for a requirement, e.g., the service response time should be below 6s. 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 = 1s for the best service response time, which is six times better as a solution with an S-goal of 6s, while under other conditions simplex may find a C-goal = 2s, which is still three times better as a solution with an S-goal.
[8]E.g., in DeltaIoT, $\mathcal{P}(k)$ are the average SNR value of different routes.

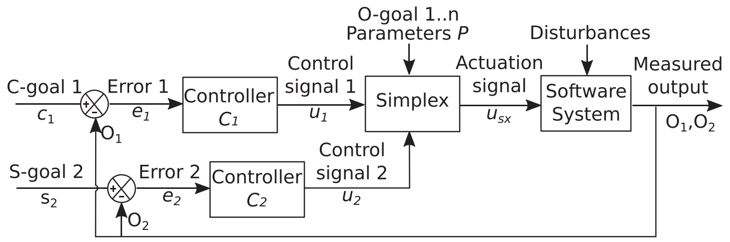


Fig. 7. Operation phase of SimCA (illustrated for one S- and C-goal).

In particular, SimCA collects all control signals $u_i(k)$ and the system parameters $\mathcal{P}(k)$ 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}) 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 Equation (1) are replaced by control signals $u_i(k)$ obtained from C_i, providing all the advantages of the controllers.

7 HANDLING UNCERTAINTY IN SYSTEM PARAMETERS

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 $\mathcal{P}$ during the Operation phase. The Parameter Updater analyzes this data using a change-point detection algorithm. When a change point is detected, the corresponding system parameter in $\mathcal{P}$ is updated to a new value (see Figure 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.

The change-point detection problem has extensively been studied in data-mining research [23, 44]. 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, such as 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 [3]. 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 handling 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 [23], 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 demonstrated by Reference [43], these two parameters provide trade-offs between time delay of change-point detection, the probability of false alarm, and the probability of correct

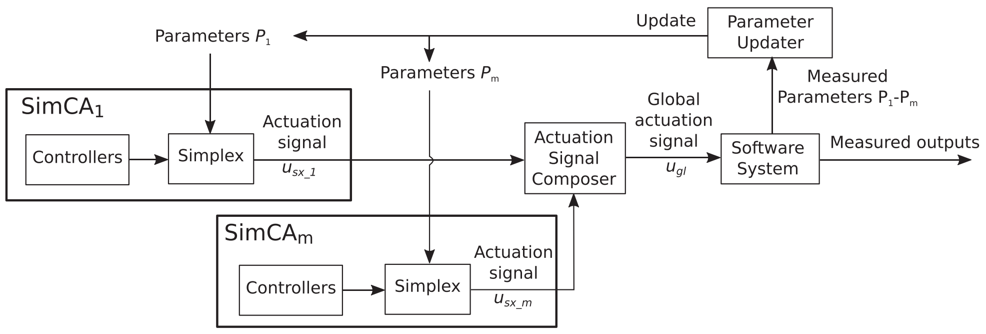


Fig. 8. Dealing with changing system parameters in SimCA*.

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 $(2 - (-2)) * 0.05 = 0.2$.

8 HANDLING UNCERTAINTY IN COMPONENT INTERACTIONS

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 its 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). 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, 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 Actuation Signal Composer should create a schedule for this components to use the resource consequently.

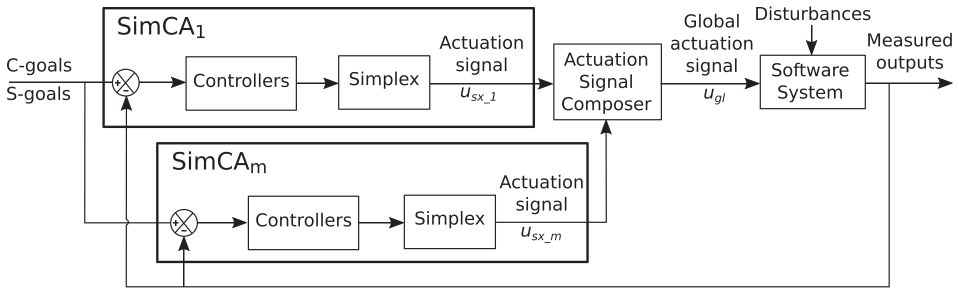


Fig. 9. Multiple instances of SimCA working in parallel.

We illustrate how the Actuation Signal Composer handles uncertainty in component interactions for the DeltaIoT case (Section 4). In DeltaIoT, the distribution of packets over different routes is calculated for all motes with multiple parents,[9] i.e., motes 7, 10, and 12 (see Figure 3). 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 3). Changing this local actuation signal to calculate the 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.

Hence, local actuation signals $u_{sx(12)}$ and $u_{sx(7)}$ are conflicting, as they both set the packet distribution probabilities over links 7-2, 7-16, and 7-3. So, the Actuation Signal Composer will first calculate the packets distributed by mote12, i.e., pd_{12}, and mote7, i.e., pd_7, based on activation probabilities of these motes and their children. Thus, 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: $u_{gl(7)} = pd_7/(pd_7 + pd_{12}) * u_{sx(7)} + pd_{12}/(pd_7 + pd_{12}) * u_{sx(12)}$.[10] In summary, the Actuation Signal Composer enables 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.

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 5) with an additional Goal Update Phase (see Figure 10).

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

[9]As all other motes have only one parent link, all packets are sent over these links.
[10]The calculation performed by the 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 [1] for details.

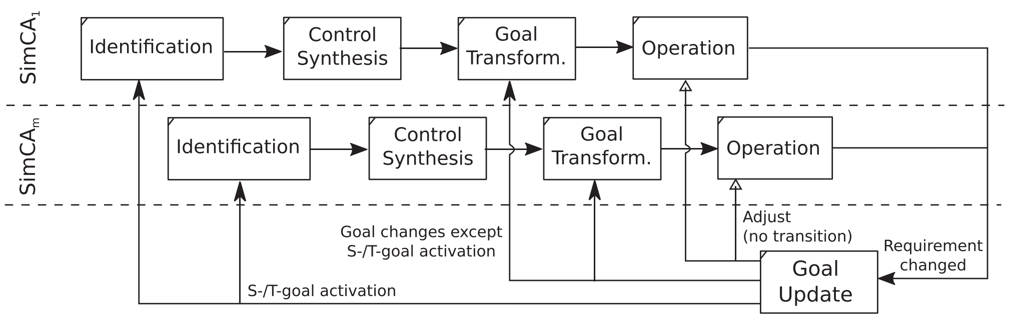


Fig. 10. Dealing with requirement changes in SimCA*.

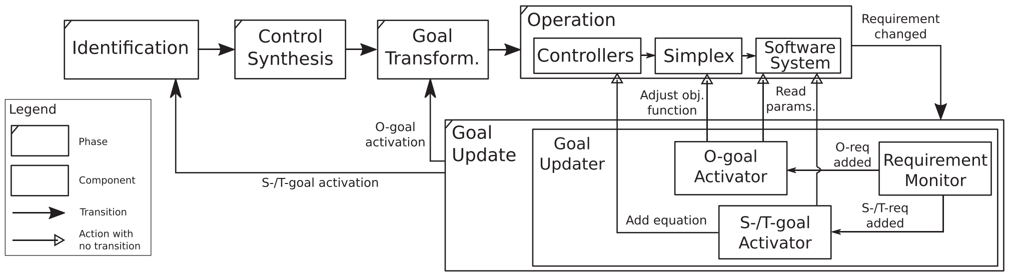


Fig. 11. Dealing with requirement activation.

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.

9.1 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 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 6.1) and reads the relevant parameters $\mathcal{P}$ related to that goal.[11] 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 6.2) 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 Equation (1) to be solved by simplex; this equation has the same structure as the equations that represent the other S-/T-goals (see Section 6.2). 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

[11]For example, if the queue loss requirement of DeltaIoT ($R3$) is activated, the Goal Activator reads the activation probabilities of all motes. We discuss details on controlling queue loss in DeltaIoT in Section 11.

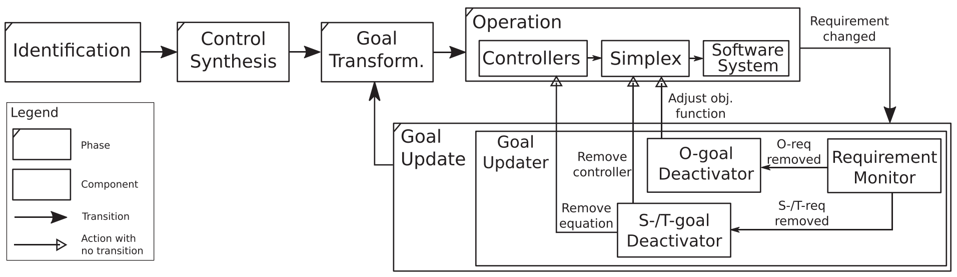


Fig. 12. Dealing with requirement deactivation.

Activator triggers Controller Synthesis to build a controller for the new goal, followed by a Goal Transformation, after which the system returns to standard Operation.

9.2 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 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.

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 requirements.

10 FORMAL EVALUATION OF GUARANTEES

SimCA* inherits a broad set of guarantees provided by basic SimCA. Since the T-goals, which are expressed as inequalities, are transformed to equalities (C-goals) during Goal Transformation (see Section 6.2), simplex that works with these equalities during Operation does not introduce additional system dynamics. Instead, it applies a straightforward translation of the control signals to an actuation signal. Furthermore, as the Actuation Signal Composer (see Figure 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 (Δe), we recall the output equation of the control system used in SimCA* [34]:

$$O_i(k) = s_i \cdot (1 - p_i^k). \tag{2}$$

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

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

In Reference [34], we derive the relation between settling time $\bar{K}$, robustness $\Delta(d)$, and pole p_i:

$$\bar{K} = \frac{\ln \Delta s_i}{\ln |p_i|} \qquad 0 < \Delta(d) < \frac{2}{1 - p_i}. \tag{3}$$

In other words, a lower value for p_i leads to weaker disturbance rejection but faster response to change. Note that in 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 infeasible 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.

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 6 [11, 12].
- Scalability: a small amount of extra time and effort are required to solve problems of growing scale. For practical problems, simplex usually finds a solution in just a few iterations [10]. This also ensures that the overhead is low for requirement changes, as only one extra simplex iteration is required.
- 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 ∞ (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 infeasible 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

Table 3. Parameters of Sensors of the UUV

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

feasible (otherwise, SimCA* will alert the user). Finally, in the current realization, SimCA* cannot provide guarantees when the system architecture is changed.

11 EXPERIMENTAL EVALUATION

We empirically evaluate SimCA* with two cases. Section 11.1 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. Section 11.2 applies SimCA* to this case and experimentally demonstrates the guarantees and quality trade-offs provided by SimCA* under different operating conditions. Section 11.3 describes the setup of the second case with DeltaIoT. In this case, adaptation is realized by SimCA* equipped with multiple basic SimCA modules. Section 11.4 shows the results of SimCA* applied to this case, while Section 11.5 compares SimCA* with a state-of-the-art architecture-based adaptation approach. In Section 11.6, we perform experiments with DeltaIoT when requirements change and a new requirement is activated at runtime. Section 11.7 demonstrates how SimCA* deals with changing system parameters at runtime. Finally, Section 11.8 discusses threats to validity. The experiments are performed on a Dell machine with a 2.7GHz Core i7 processor and 16GB 1600MHz DD3 RAM. All evaluation material is available at the SimCA* project website [1].

11.1 Experimental Setting: UUV System

First, we show the core functionality of SimCA* (Section 5) on a case of the UUV system [32]. 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, and so on, 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 five 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 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.

The UUV system has the following requirements:

- $R1$: A segment of surface scanned by the UUV when traveling over a distance of $S \geq 100$ km should be examined in $t = 10$ hours[12];
- $R2$: To perform the mission, a given amount of energy $E = 5.4$MJ is available;
- $R3$: Subject to $R1$ and $R2$, the accuracy of measurements should be maximized.

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 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 3), and uncertainty in requirements (adjustment of requirements $R1 - R3$ during operation).

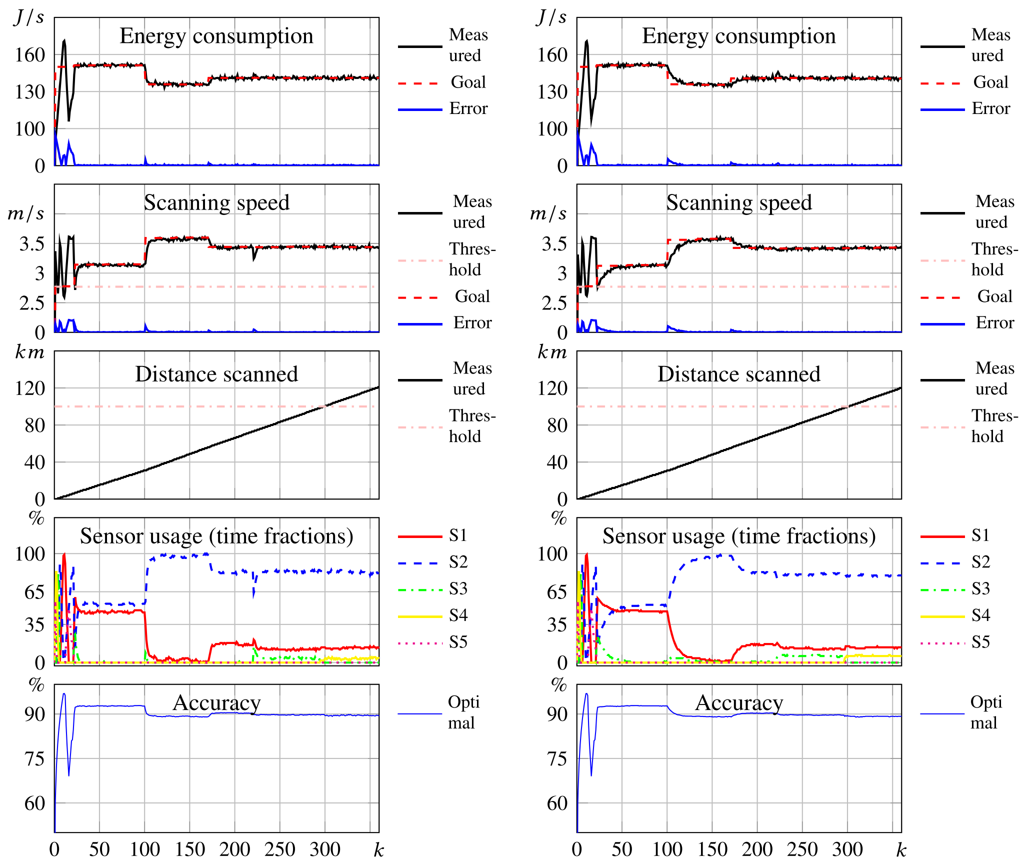
SimCA* performs adaptations every 100 surface measurements of the UUV system, i.e., the time instance k is incremented by 1 every 100 measurements. The application collects the UUV data to build performance graphs, which are used to evaluate SimCA* (see the experimental evaluation). The x-axis of the 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.

11.2 Adaptation with STO-reqs, Guarantees and Trade-offs

Figure 13(a) 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 13(a)) representing requirement $R1$ as if it was an S-req, i.e., it shows the scanning speed required to monitor exactly 100km of surface within 10h using the available energy. Since $R1$ 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 6.2). 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 $V = 3.2m/s$.

After the goal is updated, the Operation phase starts (from $k = 22$ onwards). The two upper plots in Figure 13(a) 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 5.0MJ, and at $k = 170$ from 5.0 to 5.1MJ. 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 $3.55m/s$) but decreases measurement accuracy (from 92.7% to 89.4%).

[12]When a 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 travelled distance.



(a) $p = 0.6$.

(b) $p = 0.9$.

Fig. 13. UUV adaptation with STO-reqs.

Figure 13(a) 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 $S1$ increases from 170 to $190 J/s$. To deal with this overconsumption, a portion of the time allocated to $S1$ is given to sensor $S3$, which consumes less energy (see the "Sensor usage" plot). However, at $k = 290$, $S3$ stops working and is replaced by sensor $S4$, 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 10h of time. Over a series of 50 experiments, we measured the following outcomes: the total distance scanned is 121.3 ± 0.32km, the amount of consumed energy is 5.1MJ ± 135J, the measurement accuracy is 89.94% ± 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 13(b)). After 50 runs, we got the following results: total distance scanned is 121 ± 0.28km, the amount of consumed energy is 5.1MJ ± 170J, the measurement accuracy is 89.94% ± 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 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 Δs_i chosen by the system engineer: $\bar{K} = \frac{\ln \Delta s_i}{\ln p_i}$. According to Reference [19, p. 85], the commonly used value of Δs is 0.02 (2%). Hence:

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

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 13(a) and 13(b) where the speed is required to change from 3.14 to $3.58 m/s$ (change of amplitude 0.44). Then, $\bar{K}_{0.6} = 7.66 * 0.44 = 3.4$ steps and $\bar{K}_{0.9} = 37.3 * 0.44 = 16.4$ steps. These values explain why the measured scanning speed makes almost a vertical jump at $k = 100$ in Figure 13(a), while in Figure 13(b) 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, and so on. 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 $S2$. In Figure 13(b), it is smoother and has a much lower spike at $k = 220$ than in Figure 13(b). In this case, a slower reaction may be a benefit, as it allows to switch less frequently between different sensor combinations.

11.3 Experimental Setting: DeltaIoT

We use the DeltaIoT network described in Section 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 article, 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 ~8mins for one communication cycle. In other words, using the real setup for the evaluations of SimCA* described in this article 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 the scope of this article.

All the parameter values of links and motes, including the *disturbances* of links and the *activation probabilities* of motes, are available in the Appendix. The data in these tables are based on data collected from field experiments. As explained in Section 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 8mins 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 ith goal (e.g., packet loss) during the past 10 cycles. Then it calculates the error e_i as the difference between ith setpoint (e.g., target packet loss) and the measured value of the ith 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 12h while minimizing the network energy consumption. SimCA* achieves this task by calculating the value of the *global actuation signal* u_{gl}, 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 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 12h, SimCA* performs nine 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 12h will not surpass the threshold.

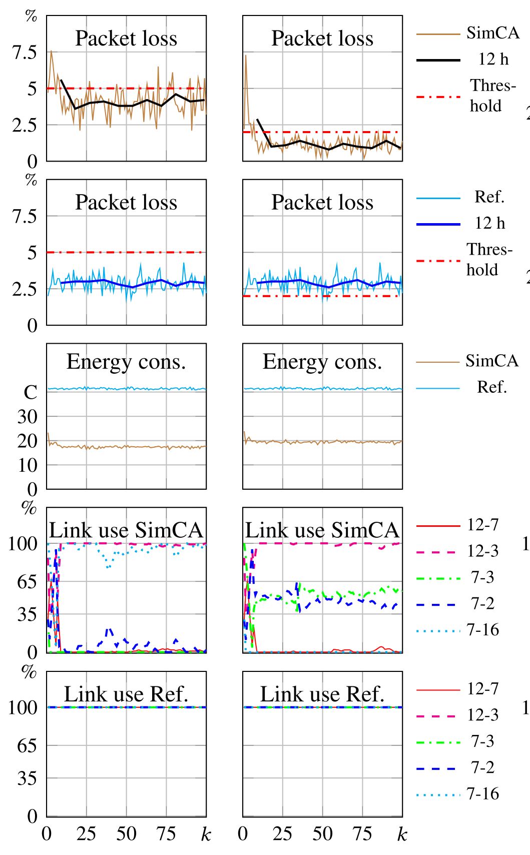
The controller pole p_i is set to 0.9, which allows to reject disturbances of high magnitude; the choice of pole values is discussed in Section 11.2. The value of δ is set at $(max_i - min_i) * 0.1$.

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.

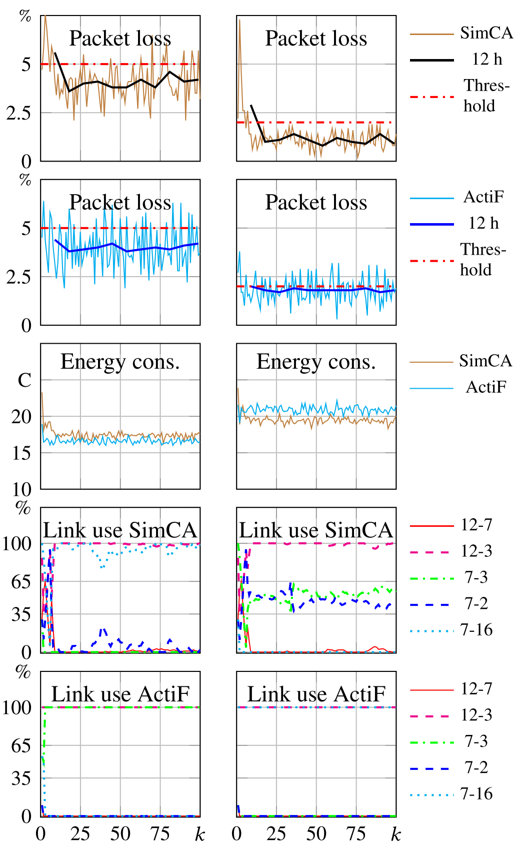
11.4 Adaptation of DeltaIoT with SimCA*

We first compare the results for packet loss and energy consumption of SimCA* and the reference approach (marked "Ref." on Figure 14(a)).[13] 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

[13]Packet loss (and queue loss) is expressed in percentages, and energy consumption is expressed in Coulomb (C in short).



(a) SimCA* vs Reference approach. Packet loss <5% (left) and <2% (right)



(b) SimCA* vs ActivFORMS. Packet loss <5% (left) and <2% (right)

Fig. 14. DeltaIoT adaptation with STO-reqs.

conservative approach is commonly used in industrial IoT networks [42]. 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 14(a) shows the results for normal operation mode, i.e., $M_{no} = \{R1, R2\}$ (Section 4). The right part of the figure shows the results with a packet loss threshold set to 2% ($R1*$), which is more challenging.

As shown in Figure 14(a), the non-adaptive reference approach is able to keep the packet loss very low during normal normal operation mode (average packet loss 2.9% ± 0.5 over $100k$), but at the cost of consuming a lot of energy (average $41.4C \pm 0.3$). 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. However, SimCA* outperforms the reference approach on energy consumption (average $16.6C \pm 0.4$ compared to $41.4C \pm 0.3$). 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 $19.5C \pm 0.6$ 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.

11.5 SimCA* vs Architecture-based Adaptation

We now compare SimCA* with ActivFORMS, a representative state-of-the-art architecture-based adaptation approach [21] that adapts the system using a Monitor, Analyze, Plan, and Execute (MAPE) feedback loop [24]. 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 [22]. 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 to provide guarantees for the adaption 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 three 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 need 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 three 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 the 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 14(b) presents the results of SimCA* and ActivFORMS. As in the previous experiments, the left part of the 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 consumption (average $17.5C \pm 0.8$ for SimCA* versus $16.6\% \pm 0.4$ for ActivFORMS). These outcomes show that for such an 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 14(b) shows that both approaches achieve the requirements, but with slightly better results for SimCA* both for packet loss (average over $K = 100$ is $1.3\% \pm 0.8$ for SimCA* versus $1.8\% \pm 0.6$ for ActivFORMS) and energy consumption (average $19.5C \pm 0.6$ for SimCA* versus $20.9C \pm 0.5$ for ActivFORMS).

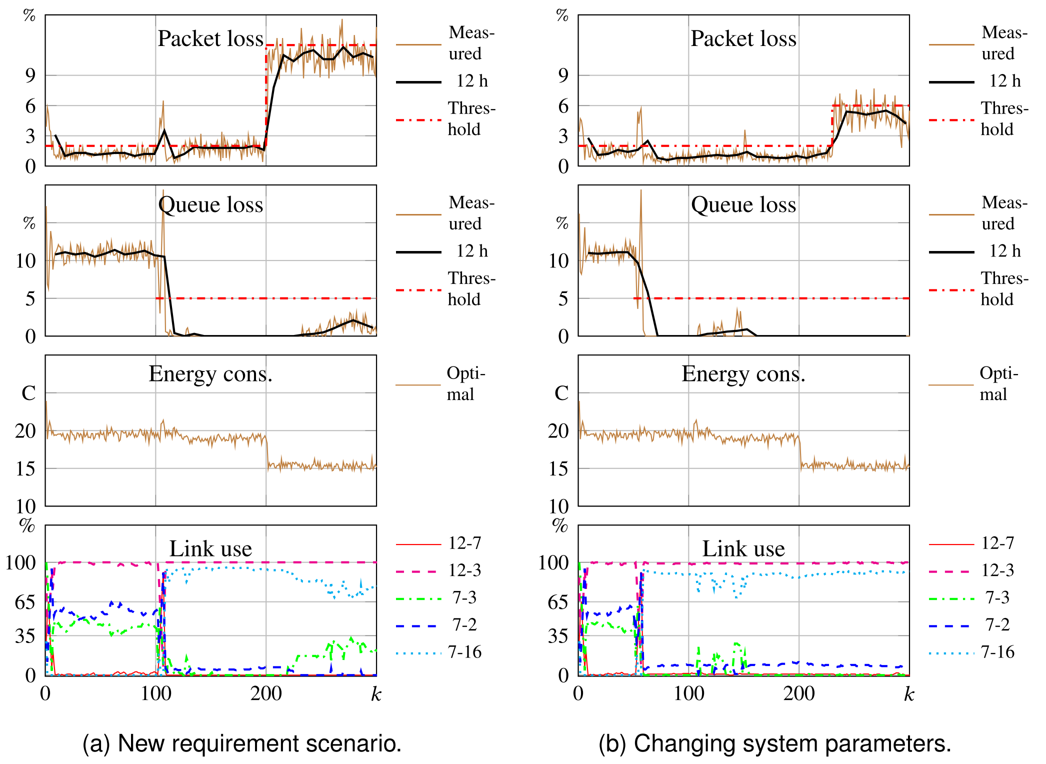


Fig. 15. DeltaIoT adaptation with changing requirements and system parameters.

The small benefits of SimCA* over ActivFORMS can be explained by the adaptation options that are available to both approaches. While SimCA (which 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. However, 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*, however, is significantly more efficient. By using simplex, the approach is able to analyze very large spaces of adaptation options to produce optimal solutions.

11.6 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 15(a) 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% ($R1*$ and $R2$). 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 $M_{bo} = \{R1*, R2, R3\}$ at $k = 100$, where requirement $R3$ (T-req) is activated. Recall that $R3$ is defined as: the average queue loss should be lower than 5% of packets sent over a period of 12h.

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 $R1*$ and $R2$ active. At $k = 100$, the new T-req $R3$ is activated (see Figure 15(a)). 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% ($R1*$) 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 15(a)) (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.

11.7 Adaptation to Changing System Parameters

To conclude, we experimentally evaluate the dynamic change of system parameters. We start from a setting with requirements $R1*$ (packet loss threshold 2%) and $R2$ (minimize energy consumption). Similarly to the experiment discussed in the previous section, we activate the queue loss requirement ($R3$), this time at $k = 50$ (see Figure 15(b)).

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 15(a), then 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 system parameters. The approach dynamically adapts to the system to deal with these uncertainties while addressing the system requirements.

11.8 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) that 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 infeasible 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 10. We also provided an initial mapping of controller guarantees to software-quality properties in Section 2.3. 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 in line with the evaluation conducted by others such as References [6, 7, 16]. However, the deployment of SimCA* in a real-world setting is required to confirm the obtained results in practice.

12 CONCLUSIONS

In this article, 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 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.

SimCA* was evaluated in a simulated distributed setting that 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 approach for self-adaptive software, which we plan to investigate in future research. Also, to confirm the obtained result in practice, we are planning to apply 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 4. SNR Values of DeltaIoT Links

	Link SNR according to power setting (0–15)																Distur-
Link	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	bance
2-4	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]
3-1	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]
4-1	−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]
5-9	−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	2.6	[−2..2]
6-4	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]
7-2	−3.0	−2.1	−1.0	0.0	0.11	1.0	1.4	2.0	2.9	3.0	4.0	4.0	4.6	5.0	5.0	5.0	[−5..5]
7-3	−7.9	−6.8	−5.9	−4.9	−4.1	−3.3	−2.4	−1.8	−0.9	−0.3	0.0	0.0	0.3	0.4	0.8	0.8	[−5..5]
7-16	−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]
8-1	−0.8	0.0	0.6	1.2	2.1	3.0	3.2	4.3	4.9	5.3	6.0	6.0	6.0	6.4	6.8	7.0	[−5..5]
9-1	−8.0	−6.8	−5.8	−4.9	−3.9	−3.0	−1.9	−1.0	0.0	0.4	1.3	2.0	3.0	3.1	3.9	4.4	[−2..2]
10-5	−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	6.0	[−5..5]
10-6	−8.1	−6.9	−6.0	−4.8	−4.0	−2.9	−2.0	−1.1	−0.1	0.4	1.2	2.0	2.7	3.0	3.9	4.0	[−5..5]
10-17	Custom SNR profile, see Figure 16																
11-7	−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	6.0	[−2..2]
12-3	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]
12-7	−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]
13-11	−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]
14-12	−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	1.0	[−5..5]
15-12	−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]
16-1	−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]
17-1	−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]

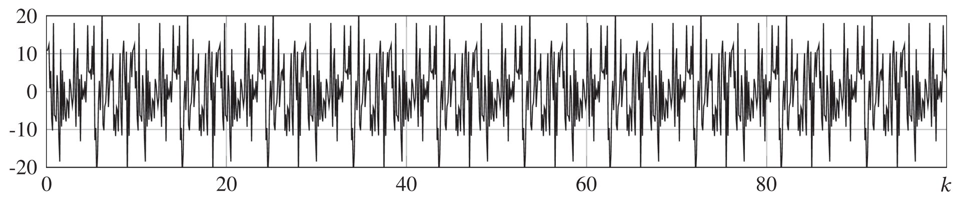


Fig. 16. Custom SNR profile of link 10-17.

Table 5. DeltaIoT Motes with Non-default Activation Probabilities

	mote5	mote7	mote11	mote12
Activation probability,%	80	80	80	90
Disturbance,%	[−10..10]	[−20..20]	[−10..10]	[−5..5]

REFERENCES

[1] SimCA* project website. 2018. Retrieved from https://people.cs.kuleuven.be/danny.weyns/software/simplex/index.htm.

[2] Ian F. Akyildiz et al. 2002. A survey on sensor networks. *IEEE Commun. Mag.* 40, 8 (Aug. 2002), 102–114.

[3] Samaneh Aminikhanghahi and Diane J. Cook. 2017. A survey of methods for time series change point detection. *Knowl. Inf. Syst.* 51, 2 (May 2017), 339–367.

[4] Konstantinos Angelopoulos, Alessandro V. Papadopoulos, Vítor E. Silva Souza, and John Mylopoulos. 2016. Model predictive control for software systems with CobRA. In *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'16)*. ACM, New York, NY, 35–46.

[5] Yuriy Brun et al. 2009. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems (Lecture Notes in Computer Science, vol. 5525)*. Springer, 48–70. DOI: 10.1007/978-3-642-02161-9_3

[6] Radu Calinescu et al. 2011. Dynamic QoS management and optimization in service-based systems. *IEEE Trans. Softw. Eng.* 37, 3 (May 2011), 387–409.

[7] Radu Calinescu, Simos Gerasimou, and Alec Banks. 2015. *Self-adaptive Software with Decentralised Control Loops.* Springer, Berlin, 235–251.

[8] Javier Camara et al. 2013. *Assurances for Self-Adaptive Systems: Principles, Models, and Techniques.* Springer.

[9] Betty H. Cheng et al. 2009. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems (Lecture Notes in Computer Science, vol. 5525)*. Springer, 1–26.

[10] George B. Dantzig. 1951. Maximization of a linear function of variables subject to linear inequalities. In *Activity Analysis of Production and Allocation.* Wiley, New York, chapter 21.

[11] George B. Dantzig and Mukund Thapa. 1997. *Linear Programming 1: Introduction.* Springer-Verlag, New York.

[12] George B. Dantzig and Mukund Thapa. 2003. *Linear Programming 2: Theory and Extensions.* Springer, New York.

[13] Rogério de Lemos et al. 2013. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II, (Lecture Notes in Computer Science, vol. 7475)*. Springer.

[14] Rogério de Lemos et al. 2017. Software engineering for self-adaptive systems: Research challenges in the provision of assurances. In *Software Engineering for Self-Adaptive Systems III. Assurances.* Springer International Publishing, Cham, 3–30.

[15] Rogério de Lemos, David Garlan, and Holger Giese. 2013. Software engineering for self-adaptive systems: Assurances, (Dagstuhl seminar 13511). Retrieved from http://drops.dagstuhl.de/opus/volltexte/2014/4508/.

[16] Ilenia Epifani et al. 2009. Model evolution by run-time parameter adaptation. In *Proceedings of the International Conference on Software Engineering.*

[17] Antonio Filieri, Henry Hoffmann, and Martina Maggio. 2014. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM, New York, NY, 299–310.

[18] Antonio Filieri, Henry Hoffmann, and Martina Maggio. 2015. Automated multi-objective control for self-adaptive software design. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering.*

[19] Joseph L. Hellerstein et al. 2004. *Feedback Control of Computing Systems.* John Wiley & Sons.

[20] M. Usman Iftikhar et al. 2017. DeltaIoT: A self-adaptive Internet of Things exemplar. In *Proceedings of the 12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'17)*. 76–82.

[21] M. Usman Iftikhar and Danny Weyns. 2014. ActivFORMS: Active formal models for self-adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*. ACM, New York, NY, 125–134.

[22] Didac Gil De La Iglesia and Danny Weyns. 2015. MAPE-K formal templates to rigorously design behaviors for self-adaptive systems. *ACM Trans. Auton. Adapt. Syst.* 10, 3, Article 15 (Sept. 2015), 31 pages. DOI: https://doi.org/10.1145/2724719

[23] Yoshinobu Kawahara and Masashi Sugiyama. 2012. Sequential change-point detection based on direct density-ratio estimation. *Stat. Anal. Data Min.* 5, 2 (Apr. 2012), 114–127.

[24] Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003).

[25] Martina Maggio et al. 2017. Automated control of multiple software goals using multiple actuators. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. ACM, New York, NY, 373–384.

[26] Sara Mahdavi-Hezavehi, Paris Avgeriou, and Danny Weyns. 2017. A classification framework of uncertainty in architecture-based self-adaptive systems with multiple quality requirements. In *Managing Trade-Offs in Adaptable Software Architectures.* Elsevier, 45–77.

[27] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2015. Proactive self-adaptation under uncertainty: A probabilistic model checking approach. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. ACM, New York, NY, 1–12.

[28] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2016. Efficient decision-making under uncertainty for proactive self-adaptation. In *Proceedings of the IEEE International Conference on Autonomic Computing (ICAC'16)*. IEEE, 147–156.

[29] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. 2008. Runtime software adaptation: Framework, approaches, and styles. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. ACM, New York, NY, 899–910.

[30] Diego Perez-Palacin and Raffaela Mirandola. 2014. Uncertainties in the modeling of self-adaptive systems: A taxonomy and an example of availability evaluation. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE'14)*. ACM, New York, NY, 3–14.

[31] William H. Press et al. 1988. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY.

[32] Mae Seto, Liam Paull, and Sajad Saeedi. 2013. Introduction to autonomy for marine robots. In *Marine Robot Autonomy*, Mae L. Seto (Ed.). Springer, New York, NY, 1–46.

[33] Stepan Shevtsov et al. 2018. Control-theoretical software adaptation: A systematic literature review. *IEEE Trans. Softw. Eng.* 44, 8 (2018), 784–810.

[34] Stepan Shevtsov and Danny Weyns. 2016. Keep it SIMPLEX: Satisfying multiple goals with guarantees in control-based self-adaptive systems. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 229–241.

[35] Stepan Shevtsov, Danny Weyns, and Martina Maggio. 2017. Handling new and changing requirements with guarantees in self-adaptive systems using SimCA*. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'17)*. IEEE Press, Piscataway, NJ, 12–23.

[36] Stepan Shevtsov, Danny Weyns, and Martina Maggio. 2018. Self-adaptation of software using automatically generated control-theoretical solutions. In *Engineering Adaptive Software Systems*. Springer Singapore. DOI: 10.1007/978-981-13-2185-6_2

[37] Vítor E. Silva Souza, Alexei Lapouchnian, and John Mylopoulos. 2012. (Requirement) evolution requirements for adaptive systems. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'12)*. IEEE Press, Piscataway, NJ, 155–164.

[38] Gabriel Tamura et al. 2013. Towards practical runtime verification and validation of self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II. (Lecture Notes in Computer Science, vol. 7475)*. Springer.

[39] Danny Weyns. 2018. Software engineering of self-adaptive systems: An organised tour and future challenges. In *Handbook of Software Engineering*, Sungdeok Cha, Richard Taylor, and Kyo Chul Kang (Eds.). Springer.

[40] Danny Weyns et al. 2012. A survey of formal methods in self-adaptive systems. In *Proceedings of the 5th International C* Conference on Computer Science and Software Engineering (C3S2E'12)*. ACM, New York, NY, 67–79.

[41] Danny Weyns et al. 2016. Perpetual assurances for self-adaptive systems. In *Software Engineering for Self-Adaptive Systems IV: Assurances, (Lecture Notes in Computer Science, vol. 9640)*. Springer.

[42] Danny Weyns, Gowri Sankar Ramachandran, and Ritesh Kumar Singh. 2018. Self-managing Internet of Things. In *Proceedings of the 44th International Conference on Current Trends in Theory and Practice of Computer Science—SOFSEM 2018: Theory and Practice of Computer Science*. Springer, 67–84.

[43] Alan S. Willsky and Harold L. Jones. 1976. A generalized likelihood ratio approach to the detection and estimation of jumps in linear systems. *IEEE Trans. Automat. Control* 21, 1 (Feb 1976), 108–112.

[44] Kenji Yamanishi and Jun-ichi Takeuchi. 2002. A unifying framework for detecting outliers and change points from non-stationary time series data. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'02)*. ACM, New York, NY, 676–681.

[45] Xiaoyun Zhu et al. 2009. What does control theory bring to systems research? *SIGOPS Oper. Syst. Rev.* 43, 1 (Jan. 2009), 62–69.

Received March 2018; revised March 2019; accepted April 2019