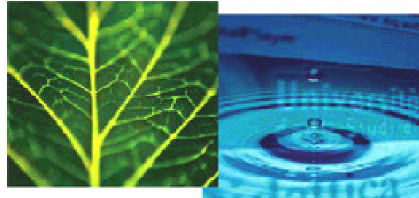
PhD Dissertation



International Doctorate School in Information and Communication Technologies

DISI - University of Trento

Optimal Adaptations over Multi-Dimensional Adaptation Spaces with a Spice of Control Theory

Konstantinos Angelopoulos

Advisor:

Prof. John Mylopoulos

Università degli Studi di Trento

April 2016

Abstract

(Self-)Adaptive software systems monitor the status of their requirements and adapt when some of these requirements are failing. The baseline for much of the research on adaptive software systems is the concept of a feedback loop mechanism that monitors the performance of a system relative to its requirements, determines root causes when there is failure, selects an adaptation, and carries it out. The degree of adaptivity of a software system critically depends on the space of possible adaptations supported (and implemented) by the system. The larger the space, the more adaptations a system is capable of. This thesis tackles the following questions: (a) How can we define multi-dimensional adaptation spaces that subsume proposals for requirements- and architecture-based adaptation spaces? (b) Given one of more failures, how can we select an optimal adaptation with respect to one or more objective functions?

To answer the first question, we propose a design process for three-dimensional adaptation spaces, named the Three-Peaks Process, that iteratively elicits control and environmental parameters from requirements, architectures and behaviours for the system-to-be. For the second question, we propose three adaptation mechanisms. The first mechanism is founded on the assumption that only qualitative information is available about the impact of changes of the system's control parameters on its goals. The absence of quantitative information is mitigated by a new class of requirements, namely Adaptation Requirements, that impose constraints on the adaptation process itself and dictate policies about how conflicts among failing requirements must be handled.

The second mechanism assumes that there is quantitative information

about the impact of changes of control parameters on the system's goals and that the problem of finding an adaptation is formulated as a constrained multi-objective optimization problem. The mechanism measures the degree of failure of each requirement and selects an adaptation that minimizes it along with other objective functions, such as cost. Optimal solutions are derived exploiting OMT/SMT (Optimization Modulo Theories/Satisfiability Modulo Theories) solvers.

The third mechanism operates under the assumption that the environment changes dynamically over time and the chosen adaptation has to take into account such changes. Towards this direction, we apply Model Predictive Control, a well-developed theory with myriads of successful applications in Control Theory. In our work, we rely on state-of-the-art system identification techniques to derive the dynamic relationship between requirements and possible adaptations and then propose the use of a controller that exploits this relationship to optimize the satisfaction of requirements relative to a cost-function. This adaptation mechanism can guarantee a certain level of requirements satisfaction over time, by dynamically composing adaptation strategies when necessary. Finally, each piece of our work is evaluated through experimentation using variations of the Meeting-Scheduler exemplar.

Keywords

[Three-Peaks, Adaptation requirements, Control Theory]

Aknowledgements

This thesis is not only a result of hard work, but also an outcome of collaborating with many people during the years I have spent as PhD student. The shared experiences, thoughts and discussions I had with them, shaped me as a researcher and I am grateful to them for their contribution to my journey in the research world.

I would like to thank my advisor, John Mylopoulos, for his guidance and for teaching me how to approach a research problem. I am also thankful for sharing his experience in research with me and for all the good quality work we produced the past four and a half years.

I am also thankful to Vitor, with whom I collaborated closely all these years and his work constituted the baseline of my research. I would also like to thank Alessandro for his contribution to the last piece of this work and sharing with me his expertise in the fascinating field of Control Theory. I owe special thanks to both of you, as well as to Martina and Julio for accepting my invitation to participate in my thesis committee.

Thanks to my colleagues at the University of Trento for the numerous discussions, brainstormings, debates and seminars we shared, trying to make each other a better researcher and person. I wish our future will bring us again together collaborating, sharing ideas, pizzas and drinks. I am also very thankful to all my friends for their company all this time, the moments of joy and the experiences we shared.

Last, but not least, I would like to thank my parents for supporting me in every decision I have taken in my life. For standing by my side whenever I was in need or I wanted to share my happiness.

To all of you, thank you, grazie mille, *ευχαριστώ*!

This work has been supported by the ERC advanced grant 267856 Lucretius: "Foundations for Software Evolution" (April 2011 - March 2016, http://www.lucretius.eu).

Contents

1 Introduction **1**

1.1 Challenges of complex software systems 2

1.2 Software system adaptation 5

1.2.1 Definitions 5

1.2.2 Feedback Loops 7

1.2.3 SISO and MIMO systems 10

1.3 Objectives of our research 11

1.3.1 Overview and contributions 16

1.4 Structure of the thesis 17

1.5 Published papers 18

2 State of the Art **21**

2.1 Baseline 21

2.1.1 Goal Oriented Requirements Engineering 22

2.1.2 GORE for self-adaptive software systems 23

2.1.3 Requirements monitoring 25

2.1.4 Variability in goal models 28

2.1.5 Requirements Evolution 29

2.1.6 Software Architecture Modelling 31

2.1.7 Software Behaviour Modelling 34

2.2 Dynamic System Modelling 34

2.3 Related Work 37

2.3.1 Requirements-based Adaptation 37
2.3.2 Architecture-based Adaptation 39
2.3.3 Behaviour-based Adaptation 41
2.3.4 Combined Model-based Adaptation 42
2.3.5 Control-based Adaptation 42
2.4 Chapter Summary 44

3 Requirements and Architecture Approaches: A Comparison 47
3.1 Selected Adaptation Approaches 48
3.1.1 Rainbow 49
3.1.2 *Zanshin* 51
3.2 The *ZNN.com* Exemplar 52
3.2.1 Overview of the problem and its architectural solution 53
3.2.2 An RE-based solution to *ZNN.com* using *Zanshin* . 56
3.3 Comparison between Rainbow and Zanshin 61
3.3.1 Methodology 62
3.3.2 Experimental Results 63
3.3.3 Discussion 65
3.4 Chapter Summary 69

4 Designing Adaptation Spaces 73
4.1 Capturing and exploring variability 74
4.1.1 Variability in behaviour 74
4.1.2 Variability in architecture 79
4.1.3 Variability in the environment 82
4.2 A Three-Peaks modelling process 83
4.3 Evaluation 87
4.4 Chapter Summary 91

5 Qualitative Adaptation for Multiple Failures 93
5.1 Requirements for Adaptation 94
5.1.1 Prioritizing Requirements 94
5.1.2 Adaptation Requirements 96
5.2 Adaptation Process for Multiple Failures 101
5.3 Evaluation 105
5.3.1 Meeting Scheduler Exemplar 105
5.3.2 Improved Adaptation 109
5.4 Chapter Summary 110

6 The Next Adaptation Problem 113
6.1 Problem Formulation 114
6.2 Prometheus Framework 117
6.3 Evaluation 120
6.3.1 The Meeting-Scheduler Exemplar 120
6.3.2 The E-shop Exemplar 126
6.3.3 Discussion 130
6.4 Chapter Summary 131

7 Control-based design of self-adaptive software 133
7.1 Model Predictive Control 134
7.1.1 Formal description 136
7.1.2 Formal guarantees 139
7.2 Design phase 141
7.3 Chapter Summary 145

8 Control-based software adaptation 147
8.1 The CobRA framework 148
8.2 Evaluation 150
8.2.1 Methodology 151

8.2.2 Experimental Results 153
8.2.3 Discussion 154
8.3 Chapter Summary 157

9 Conclusions and future work 159
9.1 Contributions to the state-of-the-art 160
9.2 Limitations of the approach 165
9.3 Future work 166

Bibliography 169

List of Tables

2.1 *EvoReqs* operations 32

5.1 Pairwise Comparison Values 95
5.2 Scale For Pairwise Comparisons 97
5.3 Differential relations elicited for the Meeting Scheduler example [SLAM13] 106
5.4 Priority Values of AwReqs 107
5.5 Evoreq operations for AwReqs 107

6.1 Control Parameter Profile. 115
6.2 Control Parameter Profile. 123
6.3 Control Parameter Profile. 127

7.1 Reference goals 143
7.2 *EvoReqs* operations 143
7.3 Indicator Priorities 144
7.4 Control Parameter weights 145

List of Figures

1.1 MAPE-K loop 7
1.2 Feedback Loop 8
1.3 SASO properties 9

2.1 Goal model for the Meeting-Scheduler case study. 24
2.2 States assumed by requirements [SLRM11]. 25
2.3 Aggregate Awareness Requirements. 26
2.4 Trend Awareness Requirement. 27
2.5 Delta Awareness Requirement. 27
2.6 Goal model for the Meeting-Scheduler case study. 30
2.7 Architectural diagram for the Meeting-Scheduler 33

3.1 The components of the *Rainbow* framework [Che08]. 50
3.2 An overview of the *Zanshin* approach [SS12]. 51
3.3 Znn.com architecture [CGS06b] 53
3.4 Strategy `SmarterReduceResponseTime` in Stitch [Che08]. . 55
3.5 Goal model for the *ZNN.com* exemplar, mirroring the adaptation scenarios modelled in *Rainbow*. 57
3.6 Specification of the `SimpleReduceResponseTime` strategy with *Zanshin*. 59
3.7 Specification of *AR3* for the `SmarterReduceResponseTime` strategy. 61
3.8 Experimental results 64

4.1 Goal model for the Meeting Scheduler case study with flow expressions. 75
4.2 BCP from AND-refinement 76
4.3 BCP from multiplicity operator 78
4.4 BCP from OR-refinement 79
4.5 ACP for component instance 80
4.6 ACP for alternative component 81
4.7 Domain model for the Meeting Scheduler environment . . . 83
4.8 The Three-Peaks process as a flowchart 84
4.9 The goal model after the Three-Peaks process 89
4.10 The architecture model after the Three-Peaks process . . . 90

5.1 Adaptation Requirements Goal Model 99
5.2 *Zanshin* Architecture 101
5.3 *Zanshin*'s Adaptation Process 103
5.4 Adaptation Requirements Goal Model [SS12] 104

6.1 Goal model annotated with contributions 115
6.2 Prometheus framework 118
6.3 Meeting-Scheduler goal model 121
6.4 E-shop goal model 128

7.1 Control scheme. 139
7.2 Meeting Scheduler goal model 142

8.1 CobRA framework 148
8.2 Indicator measured values 154
8.3 Control parameter values 155
8.4 Adaptation cost 156

Chapter 1

Introduction

> There is nothing more difficult to take in hand, more perilous to conduct, or more uncertain in its success, than to take the lead in the introduction of a new order of things.
>
> Niccolò Machiavelli

The invasion of technology in every aspect of human life places numerous challenges on Software Engineering. The growing expectations from modern software systems and their use in highly dynamic environments has resulted in founding the field of self-adaptive software systems. Such systems are meant to satisfy large sets of complex requirements in continuously changing environments, that in several cases cannot be captured before the system's deployment [EM13]. A fundamental mechanism of self-adaptive systems is the feedback loop [BSG$^+$09]. This mechanism continuously monitors if all goals are achieved and if not, a decision-making mechanism composes a new strategy to anticipate the failure. As we will see in the next chapter multiple approaches have been proposed to engineer feedback loops but very few of them propose how to design high variability self-adaptive systems that can cope with the environmental factors which lead to failures and how to manage such variability efficiently. In this chapter, we discuss the challenges that motivate the research in self-adaptive

systems. Finally, we present and overview of this thesis' proposal and contributions.

1.1 Challenges of complex software systems

As the environments of modern software systems become more dynamic, the level of uncertainty under which they operate increases dramatically. Therefore, software systems must be resilient to changes that can be *foreseen*, *foreseeable* or *unforeseen* [Lap08]. For the latter case, a configuration management [KM90] mechanism has been proposed by Kramer and Magee to respond dynamically to changes in the environment, requirements or structure of the system. The baseline of this proposal is that an external mechanism should be able to reconfigure the software system by following a change management protocol that prescribes how to add, remove, link and unlink software modules, without causing any disruptions. This mechanism should be independent of the particular application and its single objective is to maintain the system in a consistent state that is able to fulfil the system's mandate.

The increasing complexity of software systems in terms of structure and number of assigned tasks requires decision-making mechanisms that are able to compose and apply at runtime new configurations while performing trade-offs in order to achieve and maintain an equilibrium among various stakeholder goals [CGS06a]. IBM in 2001 was among the first to identify the risks of growing complexity and state the following in their technical report [Hor01]: "*As computing evolves, the overlapping connections, dependencies, and interacting applications call for administrative decision-making and responses faster than any human can deliver.*" Therefore, the required decision-making mechanisms must be part of modern software automating and improving human administration.

components must coordinate in sometimes hostile environments to achieve common goals. Due to their size, replacement of such systems is impractical in terms of cost and therefore when failures take place or requirements change, they must adapt. Another characteristic of ULS systems is that many different groups of stakeholders are involved and each of them has his or her own goals. Hence, trade-off mechanisms to satisfy all goals to a degree that corresponds to the importance of each group becomes essential.

Similar to ULS, Ubiquitous Computing [Wei93] refers to a concept of software engineering where multiple networked devices collaborate to provide various services. Applications of Ubiquitous Computing can be found in Smart Homes [EG01] and Smart Cities [CNW+12] that are designed in order to improve human daily tasks. The interconnected components of such environments may vary from smartphones, tablets to remote controlled house devices, all communicating with predefined protocols over a network. One of the main challenges in Ubiquitous Computing is that every user is unique, has his or her own goals and priorities. Therefore, the devices used by individuals in order to interact with their environment must adapt and be personalized. Moreover, user devices learn by time the habits and the preferences of the users providing them a better quality of service. Finally, as technology evolves, ubiquitous environments are populated with new kinds of devices that need to communicate with the existing ones without interrupting their operation.

Cloud computing is another emerging field where adaptation has become a necessity. This new paradigm for hosting and delivering services on-demand over the Internet poses a set of new challenges for the Software Engineering community [ZCB10]. Service providers are obliged to satisfy certain Service Level Objectives (SLOs) related to non-functional properties, known as Quality of Service (QoS). For the SLO's to be fulfilled at run-time, an automated administration mechanism is required to perform with

precision resource provisioning Virtual Machine (VM) migration in order to cope with unpredictable workload patterns and infrastructure failures. This mechanism must also balance conflicting objectives such as performance, operational cost and energy consumption and perform trade-offs that maximize the revenue of the provider and guarantee its reliability.

A new generation of systems, named Cyber Physical systems [Lee08] combine computational and physical capabilities. One of the main research challenges of this kind of systems is to guarantee a level of robustness in unknown environments by handling both software and physical failures. Given that Cyber Physical systems are destined also for mission critical operations, such as rescues in inaccessible locations, they must respond to changes and failures with high precision.

From the perspective of IT industry, multiple initiatives have provided solutions to many businesses. IBM's Autonomic Computing presented in 2001 by Horn [Hor01] has been a point of reference for both academic and industrial research on the topic. This work is followed by Sun's N1 management software [Sun], that was designed to tackle to problem of managing large, complex and heterogeneous infrastructures. Microsoft with its Dynamic Software Initiative [Mic] contributed to a cost-effective automatic resource allocation in order to meet the growing demands of the market. At the same time Hewllett-Packard introduced the Adaptive Enterprise Strategy approach while Intel proposed standards for implementing autonomic computing solutions [TM06].

Despite the industrial efforts for producing software that can successfully operate in the modern ever-changing environment as new technological applications arise, software engineers must tackle more challenges. Therefore, general principles for designing such systems in order to remain sustainable through time are necessary. Moreover, as the field of software adaptation becomes more interdisciplinary, software engineering practises

must include techniques and guidelines from other fields such as Control Theory, Mathematical Optimization, Artificial Intelligence, Formal Methods and others, in order implement effective decision-making and planning mechanisms.

1.2 Software system adaptation

A solution proposed for dealing with the increasing complexity of software systems and the uncertainty of their environment is to develop systems that can manage themselves while being aware of the goals that they must fulfil. This section describes the fundamental concepts that have been the baseline of our work and provides the necessary definitions about the properties the examined systems must demonstrate [1].

1.2.1 Definitions

In the literature the terms *self-adaptive* and *autonomous* system are often used interchangeably. However, according to Huebdcher and McCann [HM08] self-adaptive systems are a subset of autonomic systems, whereas McKinley et al. citemckinley2004composing argue that self-adaptive has less coverage as it refers mostly to applications and middleware as opposed to autonomic systems that handle all layers of the system's architecture. Laddaga and Robertson [RL05] use the definition of self-adaptive software that was provided by a DARPA Broad Agency Announcement on self-adaptive software (BAA-98-12) in December of 1997 and we adopt through this thesis:

> Self-Adaptive Software evaluates its own behaviour and changes behaviour when the evaluation indicates that it is not accom-

[1] This thesis extends the work of Vitor E.S. Souza [SS12] and therefore, shares a certain number of definitions and research baseline.

> plishing what the software is intended to do, or when better functionality or performance is possible. [...] This implies that the software has multiple ways of accomplishing its purpose, and has enough knowledge of its construction to make effective changes at runtime. Such software should include functionality for evaluating its behaviour and performance, as well as the ability to replan and reconfigure its operations in order to improve its operation.

On the other hand, the *adaptive* software is identical to self-adaptive software except from the fact that the first one delegates to external actors the decision-making process about the new configuration that must be applied. A common case of such systems are socio-technical systems [Bry09], where humans are involved in the loop of the adaptation process.

In IBM's Vision of Autonomic Computing [KC03] it is presented a set of properties that must characterize each self-adaptive system. These properties are referred as as self-* properties and are described below:

- **Self-configuration.** The configuration of the components of the system should be automated and follow a set of high-level policies. The rest of the system must adjust automatically to the new configuration.
- **Self-optimization.** The components of the system constantly seek to optimize and improve the performance and efficiency of the overall system.
- **Self-healing.** The system automatically detects failures, diagnose their cause and take actions to restore the malfunctioning software or hardware.
- **Self-protection.** The system must be able to defend and recover from malicious attacks. Hence, the system must have the capability to compose plans in order to anticipate such attacks.

1.2.2 Feedback Loops

The paradigm proposed by IBM for engineering self-adaptive systems involves the adoption of a basic concept from Control Theory, the feedback loop [BSG+09]. More specifically, a self-adaptive system must perform a set of actions in order to guarantee the aforementioned self-* properties. This loop is depicted in Figure 1.1 and is composed of the following actions:

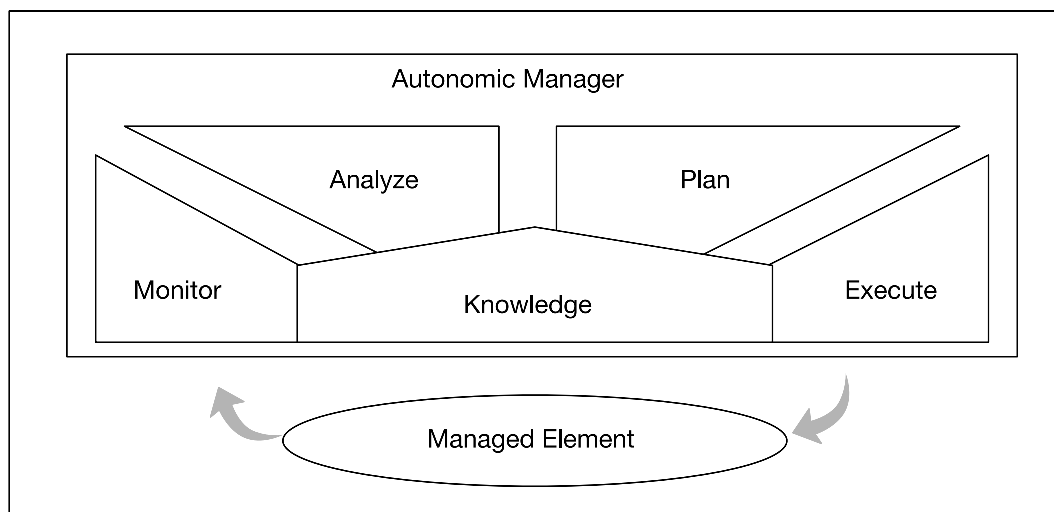


Figure 1.1: MAPE-K loop

1. **Monitor.** A set of sensors capture event data from the managed element's operation and its environment. Then, the collected data is registered in a knowledge base for future use.

2. **Analyze.** The analyzer compares the most recently received data with the existing patterns in the knowledge base and diagnoses failures in the managed element and their symptoms.

3. **Plan.** The planner, based on the cause of failure, composes a plan that will lead the managed element to recovery.

4. **Execute.** A set of effectors interpret the high level adaptation plan to low level actions and apply the changes to the managed element.

Despite the fact that the Autonomic Manager that is responsible for performing these actions is presented as an external mechanism to the managed element, this distinction is more conceptual rather than architectural.

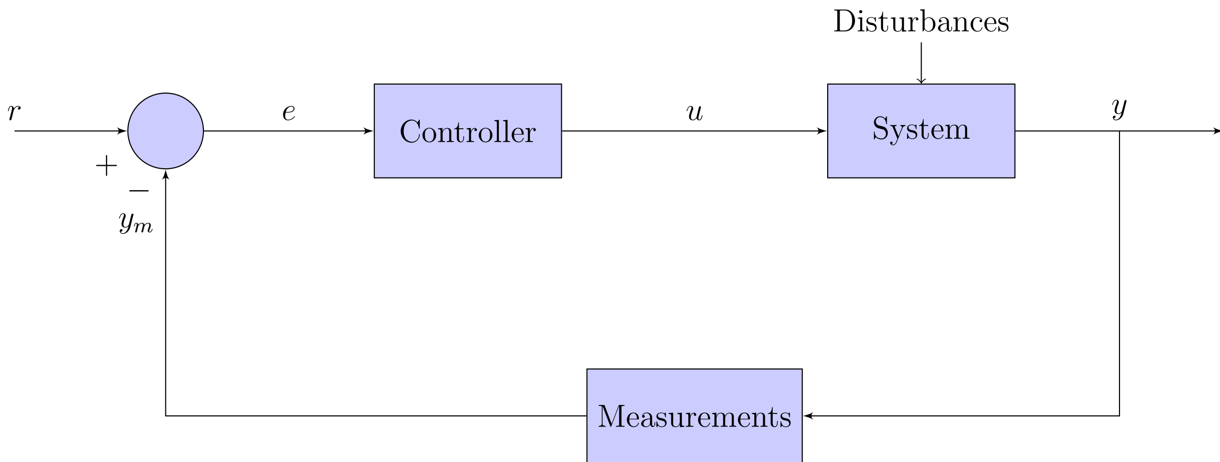


Figure 1.2: Feedback Loop

From a control engineering point of view, a feedback loop is constructed as presented in Figure 1.2. The reference input (r) is the desired value of an elicited measurable goal. The output of the system (y) is measured by sensors and in several cases is also filtered. The measured and filtered output (y_m) is compared to the reference input and their difference is known as *control error*. The controller receives as input the control error and changes values of control parameters in order for the measured value to converge to the desired one. The adaptation process and in particular the controller, must demonstrate certain characteristics known as SASO (stability, accuracy, settling time and overshoot) properties [HDPT04] depicted in Figure 1.3 and explained below:

- **Stability.** The output of a stable system must always converge to a desired value, given by a reference input. Despite the fact that

the convergence is not constant due to disturbances from the environment, there are operating regions (i.e. combinations of workloads and configuration settings) in which their performance is considered acceptable.

- **Accuracy.** This property refers to how close the measured output converges to the desired value. Ideally, the measured value should be equal to the desired value.
- **Settling time.** This refers to the time it takes to the controller in order to drive the system's goal as close as possible to the reference input and must be minimal.
- **Overshooting.** This property refers to the maximum difference between the measured value and the desired value.

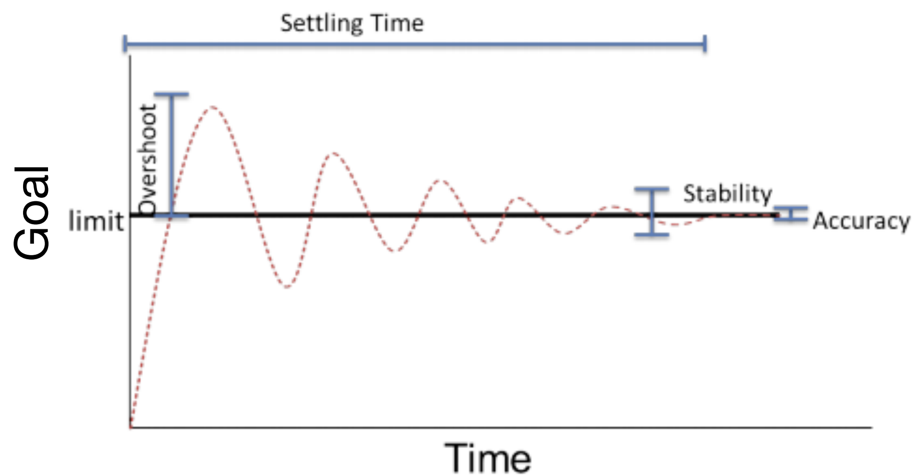


Figure 1.3: SASO properties

An additional property of equal importance is robustness. A controller bases its decisions on a model that describes the system's dynamics. In the particular case of software systems there are no laws of nature that could provide such models. Therefore, the dynamics of the system can be derived

through a process named system identification [Lju99] that provides approximate models of the system's behaviour. Moreover, the measurements in the outputs might be biased and often inaccurate. A robust control system is capable of overcoming such inaccuracies and converge to the desired value of its goal.

1.2.3 SISO and MIMO systems

Systems where there is just one control parameter the value of which is decided by the controller and one output are called Single Input Single Output systems. Consider a news website that is hosted by a number of replicated servers. The servers are not property of the news website but are rented and can be allocated and released dynamically. When a popular article is posted on the website the traffic increases dramatically and more servers must be allocated in order to maintain the desired response time. Hence, while the system operates the controller must decide the number of servers that are required to satisfy the connected clients. The number of servers influences the output of the system and can be tuned by the controller, hence it is a control parameter. Control Theory has provided solutions such as the Proportional-Integral-Derivative (PID) controller [Ast95] that if designed properly can demonstrate all the aforementioned properties.

Unfortunately, most systems in the software world are more complex, including a large set of control parameters and outputs that occasionally are coupled to each other. In the previous example, response time is not the only output for which the stakeholders provide requirements. The servers are rented and therefore, they bear a certain operation cost for the service provider of the news website. An alternative to increasing the number of servers in order to reduce the response time could be to reduce the resolution of the multimedia content that is hosted on the website, which though is going to decrease the fidelity of the users that is prescribed

by the stakeholders to remain high. One can easily understand that as the number of control parameters, hereafter referred as *adaptation space*, and the number of monitored goals grow, the self-optimization property becomes increasingly more challenging. In this thesis we discuss only the second category of systems and we refer to them as Multiple Input Multiple Output (MIMO) systems [SP07].

1.3 Objectives of our research

In Section 1.2 we have described the main challenges in the field of self-adaptive software systems and the basic concepts that define the research direction for their design and implementation. We now specify explicitly the research objective of this thesis, what are the open research questions that we address to and present an overview of their answers.

Research Objective: *to design high variability self-adaptive systems that combine control parameters from their requirements, architecture and behaviour and develop adaptation mechanisms capable of dealing with multiple failing requirements and making optimal decisions wrt the priority of each failure.*

RQ1: How does an adaptation space based on requirements relates to architecture-based adaptation spaces?

In the beginning of our research, we investigated similarities and differences between approaches that use requirements models for software adaptation and others that use architecture models. The reason is that, as we will see in the next chapter, requirements and architectural approaches for designing self-adaptive systems cover the largest portion of the literature in the

field. This triggered our comparison study, where we used the same exemplar and applied an representative framework from each category. The main difference was found to be that requirements-based approaches capture high level goals and usually ignore the capabilities and the restrictions of the target system, since those become available later, when design decisions are taken. On the other hand, architecture-based approaches focus on lower level requirements and are aware of the technical limitations of the system. The conclusion of our comparison is that a combination of the two approaches would capture in detail essential aspects of the software system.

RQ2: Can we extend existing techniques to relate requirements-based adaptation spaces to other aspects of software systems?

To answer this question it is important to identify what are the variability dimensions of a self-adaptive software system and how these are related to each other. Variability is essential to self-adaptive software systems, because it captures the space of alternative adaptations a system is capable of applying to cope with changes in its environment. Our work goes beyond the existing one-dimensional view of adaptation spaces by defining adaptation spaces that accommodate three complementary dimensions. The first dimension captures variability in fulfilling requirements and represents variability in the problem part of the adaptation space. The other two dimensions capture variability with respect to behaviour and architecture. These dimensions capture variability in the solution space of the system-to-be, representing *how*, by *whom* and in *what* sequence requirements are to be fulfilled. The variability of these dimensions is captured by a process named *Three-Peaks*, by guiding designers to define iteratively an adaptation space by introducing some requirements, deciding on their architectural and behavioural dimensions, and then going back and intro-

ducing more requirements, including ones that are determined by architectural and behavioural decisions. This work extends the Twin-Peaks approach [Nus01] that intertwines software requirements and architectures promoting incremental development for faster specifications.

RQ3: How do we deal with multiple failing requirements under the absence of quantitative information that describe the system dynamics?

One of the main challenges in the area of self-adaptive software systems is that there are no laws of nature to describe their behaviour. Therefore, the impact of changing one control parameter from the adaptation space on one or more systems goals is not known a priori. This increases the complexity of the decision-making process. The reason is simple: in case of failures F, F', the candidate adaptations A, A' may be conflicting, as A may call for a behaviour that exacerbates F', and vice versa with A'.

We tackle this problem by prioritizing systematically software requirements with the use of the Analytical Hierarchy Process (AHP) [KR97]. The priorities are used in cases where the adaptation mechanism when multiple failures are present and conflicts among requirements are present. In such cases, the most important failures with respect to their priority are selected to be fixed. Then we define a new kind of requirements named *Adaptation Requirements* (*AdReqs*) provided by stakeholders that prescribe policies and constraints for the adaptation process itself. For example, an adaptation requirement may state that the adaptation should be conservative in that it does not change parameters in a way that could harm non-failing requirements. We also propose a qualitative adaptation process that takes into account adaptation requirements and iteratively collects failures, selects an adaptation, applies it, and observes results.

RQ4: How could the self-adaptation problem be formulated as an optimization problem and how could it be solved?

For the cases in which quantitative relations between control parameters and software goals are available multi-objective optimization techniques can be applied for selecting an optimal adaptation. In our work, there are two criteria that such an adaptation must satisfy: a) minimize the degree of failure, (i.e. the control error we presented in Section 1.2.2) for system requirements with respect to their importance and b) optimize lexicographically [Ise82] quality attributes (e.g. cost, performance etc.) of the system.

Before selecting an adaptation it is important to locate the cause of failures. This means that the adaptation space is dynamic and the available solutions depend on the failures that caused them. For instance, when a Meeting-Scheduling system fails to book rooms, one possible solution is to dispose more rooms. However, this might not be effective, because the cause of failure was a long downtime of the external service that is responsible for finding and booking rooms. Therefore, identifying the cause of failure is critical for choosing an effective adaptation. Moreover, software systems are characterized of various kinds of dependencies i.e. a change in one control parameter enforces a change to another one.

The formulation we propose consists of three steps. First, goals and quality attributes of the system are prioritized in the same manner as we did for answering **RQ3**. Second, we define an objective function that measures the aggregated degree of failure of each requirement and its independent variables are the control parameters of the system. Finally, we define the constraints that capture the various dependencies of system goals and components, the boundaries of the control parameters and ex-

clude solutions that are not effective based on the root cause of failures. Finding values for the control parameters in order to minimize the defined objective function is referred as the *Next Adaptation Problem*.

This problem rises every time one or more requirements of the system fail. As a solution to the Next Adaptation Problem we propose a framework that monitors the success of system goals and when failures are detected a root cause analysis component identifies the source(s) of failure. Based on the output of this component a new adaptation space is composed with all the candidate solutions for the occurred failure. Then an optimization component finds values for the available control parameters that minimize, ideally eliminate, the failures and optimize lexicographically the system's quality attributes.

RQ5: How to find an optimal adaptation under the absence of any information about system's dynamics?

The solution to the previous question is based on the assumption that quantitative information is available by domain experts. Given the quick pace new kinds of application are introduced such expertise cannot be taken for granted.

We tackle this problem with the use of Control Theory and more specifically Model Predictive Control [CBA04]. Our approach integrates software development with control engineering practises by simulating the system-to-be and eliciting an analytical model that captures the relation between goals and the success rate of the monitored goals. A controller uses this model in order to predict the system's behaviour and make any necessary changes in order to maintain the control error of each goal to the minimum with respect to its priority.

The controller is part of a framework that monitors success rates of

functional and non-functional requirements and when control errors occur the embedded controller composes an adaptation plan to minimize them. Inevitable inaccuracies and nonlinearities of the analytical model are handled by a Kalman filter [Lju99] that linearizes the model at runtime over an operational point. Furthermore, Model Predictive Control can provide formal guarantees for satisfying the SASO properties that we discussed earlier.

1.3.1 Overview and contributions

In summary the contributions of this thesis are:

- A systematic process — Three-Peaks — for extracting incrementally variability from goal models. We model requirements, behavioural and architecture control parameters as well as parameters of the environment. The purpose of this process is to derive a sufficiently large adaptation space, able to cope with environmental uncertainty responding to **RQ1** and **RQ2**.
- A new type of requirements — *AdReqs* — that capture constraints of the adaptation process itself. This new type of requirements is meant to increase the precision of the proposed adaptation mechanisms and respond to **RQ3**, **RQ4** and **RQ5**.
- A qualitative adaptation mechanism that exploits requirement priorities in order can handle multiple failures without any analytical models for the system's dynamics. This also contributes to **RQ3**.
- A formulation of the Next adaptation problem and a framework that exploits quantitative models, root cause analysis to solve it. This addresses **RQ4**.

- A set of guidelines for applying control engineering practises in the development of self-adaptive software for eliciting the system's behaviour and design a controller that can correct multiple requirement failures offering formal guarantees. This addresses **RQ5**.

1.4 Structure of the thesis

The remainder of this thesis presents in detail the proposed approach we summarized above in the following structure:

- Chapter 2 overviews the research baseline of our proposal and the state-of-the-art.
- Chapter 3 presents a comparison between two model-based adaptation mechanisms. One uses architecture and the other requirements models. The results of this comparison reveals the advantages and disadvantages of each approach.
- Chapter 4 describes the sources of variability in software system design and proposes models to capture requirement, behavioural, architectural and environmental variability. Moreover, it describes a systematic iterative process that guides the elicitation of this multi-dimensional variability.
- Chapter 5 presents the concept of Adaptation Requirements and a qualitative adaptation mechanism for handling multiple failures.
- Chapter 6 defines the problem of adaptation as a constrained multi-objective optimization problem and describes in detail a framework that performs root cause analysis each time one or more requirements fail, composes a new adaptation space and composes an optimal adaptation.

- Chapter 7 describes how the design of an MPC controller can become part of software engineering for self-adaptive systems.
- Chapter 8 presents a framework that uses an MPC controller to produce adaptation plans by exploiting estimated analytical models that describe the system's dynamics.
- Chapter 9 concludes the thesis with a summary of our contributions, discussing the advantages and the limitations of our proposal as well as the possibilities for a new research agenda.

1.5 Published papers

- Vítor E. Silva Souza, Alexei Lapouchnian, Konstantinos Angelopoulos, John Mylopoulos: Requirements-driven software evolution. *Computer Science - R&D 28(4): 311-329 (2013)*
- Konstantinos Angelopoulos, Vítor E. Silva Souza, João Pimentel: Requirements and architectural approaches to adaptive software systems: a comparative study. *SEAMS 2013: 23-32*
- João Pimentel, Konstantinos Angelopoulos, Vítor E. Silva Souza, John Mylopoulos, Jaelson Castro: From Requirements to Architectures for Better Adaptive Software Systems. *iStar 2013: 91-96*
- João Pimentel, Jaelson Castro, John Mylopoulos, Konstantinos Angelopoulos, Vítor E. Silva Souza: From requirements to statecharts via design refinement. *SAC 2014: 995-1000*
- Konstantinos Angelopoulos, Vítor E. Silva Souza, John Mylopoulos: Dealing with multiple failures in zanshin: a control-theoretic approach. *SEAMS 2014: 165-174*

- Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolás D'Ippolito, Ilias Gerostathopoulos, Andreas B. Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro Vittorio Papadopoulos, Suprio Ray, Amir Molzam Sharifloo, Stepan Shevtsov, Mateusz Ujma, Thomas Vogel: Software Engineering Meets Control Theory. *SEAMS@ICSE 2015: 71-82*

- Konstantinos Angelopoulos, Alessandro Vittorio Papadopoulos, John Mylopoulos: Adaptive predictive control for software systems. *CTSE@SIGSOFT FSE 2015: 17-21*

- Konstantinos Angelopoulos, Vítor E. Silva Souza, John Mylopoulos: Capturing Variability in Adaptation Spaces: A Three-Peaks Approach. *ER 2015: 384-398*

- Konstantinos Angelopoulos, Fatma Basak Aydemir, Paolo Giorgini, John Mylopoulos: Solving the Next Adaptation Problem with Prometheus. *RCIS 2016*

- Konstantinos Angelopoulos, Alessandro Vittorio Papadopoulos, Vítor E. Silva Souza, John Mylopoulos: Model Predictive Control for Software Systems with CobRA. *SEAMS 2016 (accepted)*

Chapter 2

State of the Art

> To know what you know and what you do not know, that is true knowledge.
>
> Confucius

The area of self-adaptive software is broad and interdisciplinary with rich literature of diverse approaches that tackle the problem of software adaptation using a variety of conceptual models and techniques. In this chapter we overview the baseline this thesis is built on and we summarize the state-of-the-art in this area.

2.1 Baseline

Every software development project starts with the elicitation of stakeholder requirements and continues with making decisions on the design of the system-to-be in order to satisfy as many as possible of these requirements. The particularity of self-adaptive systems is that their requirements might change over time or cease to be satisfied due to the environment's uncertainty. Our work treats requirements as first class citizens and the point of reference when a new adaptation plan must be composed. The next sections introduce concepts related to Requirements Engineering and Software Variability along with their applications in software adaptation.

2.1.1 Goal Oriented Requirements Engineering

The importance of requirements in software engineering development is introduced by Ross and Schoman [RS79] with the notion of the requirements problem. More specifically, they stated that: 'even the best structured programming code will not help if the programmer has been told to solve the wrong problem, or, worse yet, has been given a correct description but has not understood it.' Later, Zave and Jackson [ZJ97] suggest that the requirements problem amounts to finding the specification **S** that for given domain knowledge **K** satisfies the given requirements **R**. The previous sentence is depicted in the in the following mathematical logic form $\mathbf{K}, \mathbf{S} \vdash \mathbf{R}$.

Goal-Oriented Requirements Engineering (GORE) was founded with the premise to give a solution to the requirements problem with the use of modelling languages. According to GORE, stakeholder requirements are expressed as goals — desired state-of-affairs — which must be elicited, modelled, analyzed and validated. There is rich literature [vL00] which focuses on producing requirement specifications that includes these steps.

In their work Jureta et al. put the requirements problem in the context of adaptive systems [JBEM14], advocating the need of configurable specifications. Such specifications prescribe a set of possible configurations for the system-to-be that only one is active at runtime. Therefore, when the initial assumption about the system or the requirements change a new configuration is applied. The work by Souza [SS12], which this thesis extends, addresses to this problem by proposing requirements to monitor the success or failure of other requirements and capture changes at requirements level. In the same line of work, a qualitative adaptation mechanism is proposed and performed by the *Zanshin* framework [SS12] which we explain in detail in the next Chapter.

2.1.2 GORE for self-adaptive software systems

Goal Models. In this thesis we also use goal models for representing stakeholder requirements. A goal model captures both functional and non-functional requirements, referred as *hard goals* and *soft goals* respectively. Hard goals are *AND/OR*-refined until each goal is operationalized by tasks. Along with goals, *domain assumptions* represent preconditions that must hold for the system to operate properly.

Figure 2.1 captures the requirements of a Meeting-Scheduler system [LDM95] that is meant to facilitate the process of organizing meetings in a large institution. For the system to satisfy its top goal ScheduleMeeting, every time a meeting request arrives, must initiate a meeting by creating a new meeting event, collect the participant list, the meeting's topic and the required equipment for the meeting. Next, the timetables are collected by each participant either by e-mail, by phone or automatically by the system. In order for the system though to collect the timetables the domain assumption that participants use the system calendar must hold. Then, a date and room must be selected either manually by the meeting organizer or automatically by the system. In addition, the system must allow the meeting organizer to confirm or cancel the occurrence of the meeting, send invitations to the participants and modify the date or the topic if needed. Finally the system must store all the date related to the meeting and make them accessible to the meeting organizer.

Soft goals represent desired qualities of the system-to-be [MCN92]. In spite of their qualitative nature, soft goals can be operationalized by *quality constraints* that quantify the degree to which they are fulfilled. For example, *Fast Scheduling* may be operationalized by the quality constraint duration(Schedule Meeting) $\leq$ 6hrs. Alternatively, quality goals can be operationalized by optimization constraints, named *quality attributes*. For

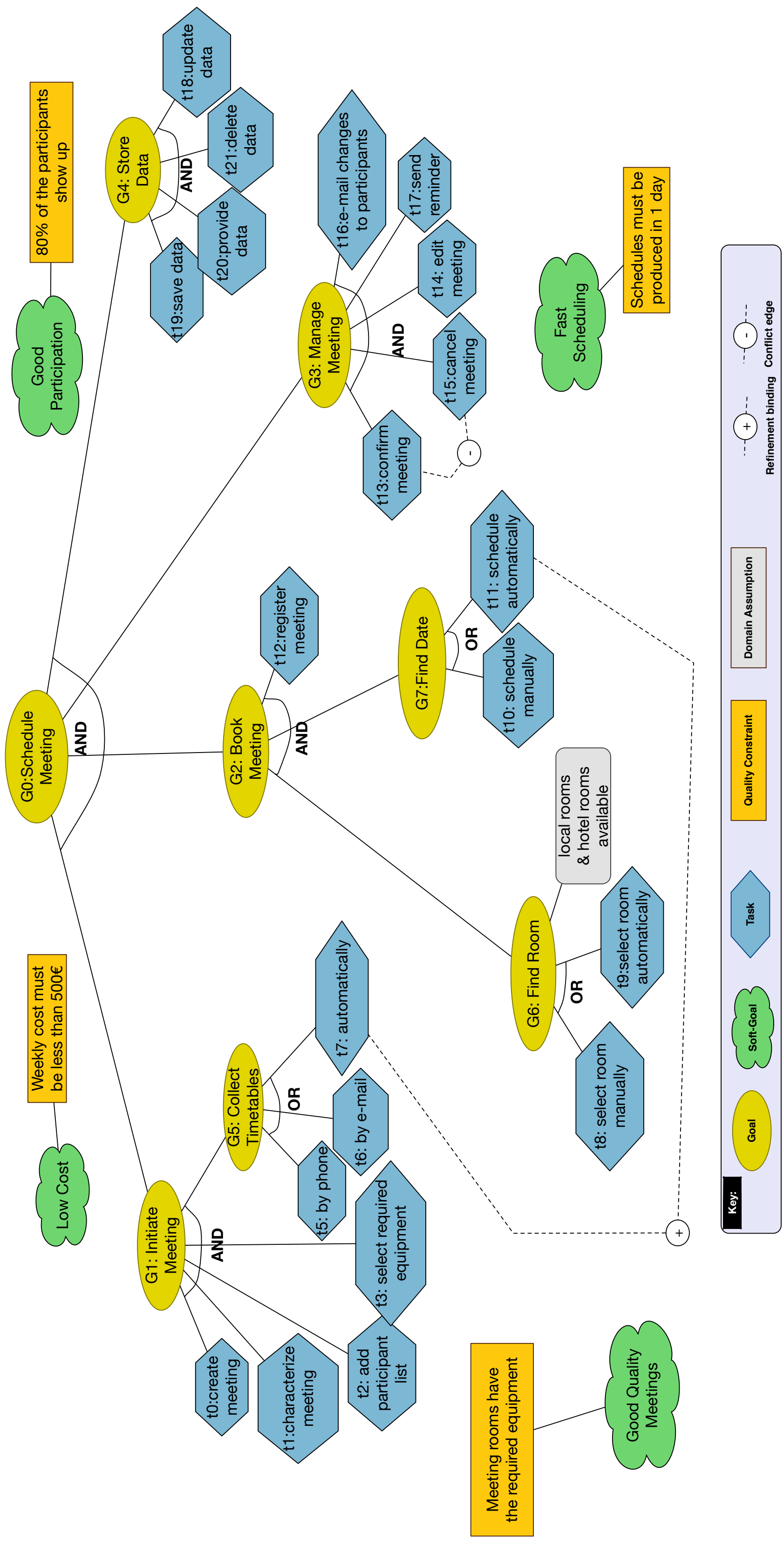


Figure 2.1: Goal model for the Meeting-Scheduler case study.

instance, *Fast Scheduling* may be operationalized by minimizing the time it takes to schedule meetings.

A goal model for a self-adaptive system captures the functional requirements for the system-to-be. The system at runtime can switch among alternative refinements where this is possible, in order to guarantee the satisfaction of all root goals. However, choices among alternatives can be constrained. For instance, as Figure 2.1 shows, if the timetables are collected automatically then the meeting's date must be selected automatically by the system as well. Such relationships are called *goal constraints* [NSGM16a] and capture dependencies among goals.

2.1.3 Requirements monitoring

As explained in the previous Chapter, one of the fundamental components of an adaptation mechanism is monitoring. A system must be aware of its goals and the degree in which they are fulfilled at runtime [SBW+10]. Fickas and Feather in their work [FF95] explore the need of specifications with focus on monitoring requirements of systems that operate in ever-changing environments.

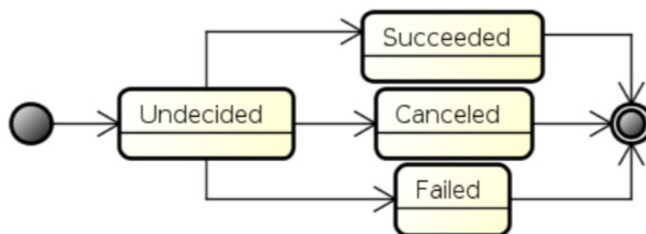


Figure 2.2: States assumed by requirements [SLRM11].

The requirements of a software system are meant to be satisfied more than once at runtime. In other words, one can assume that each elicited requirement is a class that is going to be instantiated multiple times. For

example, the goal CollectTimetables of the Meeting-Scheduler exemplar must be fulfilled every time a new request arrives and therefore, a new instance of this goal is created. Goal instances go through certain states as shown in Figure 2.2. When the goal instance is created its state is Undecided. Eventually as the system pursues to fulfil the goal, the instance will either have Succeeded or Failed. In case the goal is taking too long to be fulfilled, another potential state is Cancelled.

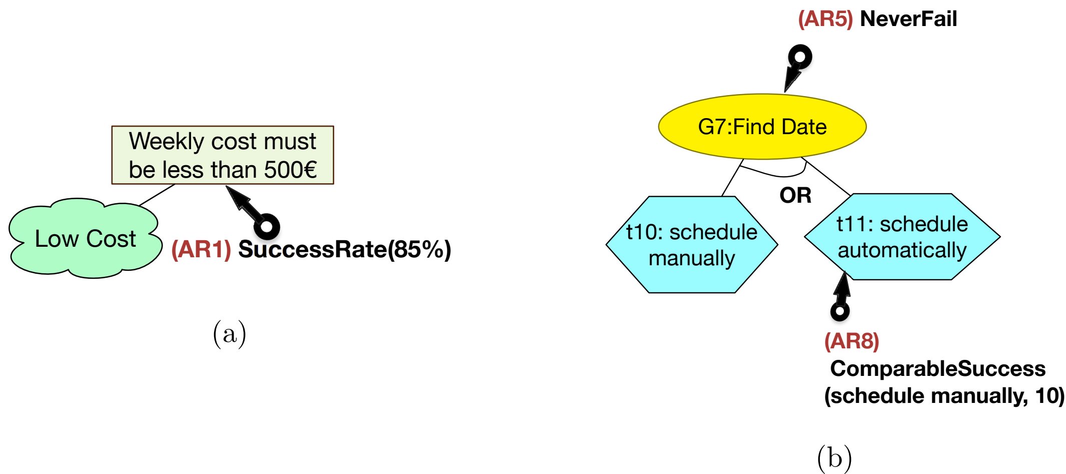


Figure 2.3: Aggregate Awareness Requirements.

Awareness Requirements (*AwReqs*) are associated with goals, tasks, domain assumptions and quality constraints. The monitoring mechanism, every time a new instance of a goal is created records every change in its state. This allows to measure the success of requirements over time. For example, the quality constraint of the soft goal Low Cost prescribes that the weekly cost of meetings must be less than 500€. In Figure 2.3a the *AwReq* $AR1$ prescribes that the 85% of its instances must have been in the state Succeded before the end of their sessions, which in this case lasts one week. Another example is AR5 that indicates none of the instances of the goal Find Date must ever be in the state Failed, whereas $AR8$ that the task schedule automatically succeeds ten times more than the task sched-

ule manually. Such *AwReqs* that monitor the states of other requirements over the time are referred as aggregate *AwReqs*.

Trend *AwReqs* is yet another type of *AwReq* that compares success rates over a number of periods. For example, in Figure 2.4 *AR*6 prescribes that the success rate of the goal Collect Timetables should not decrease two weeks in a row. This type of *AwReqs* is used to identify how success/fail rates evolve over time.

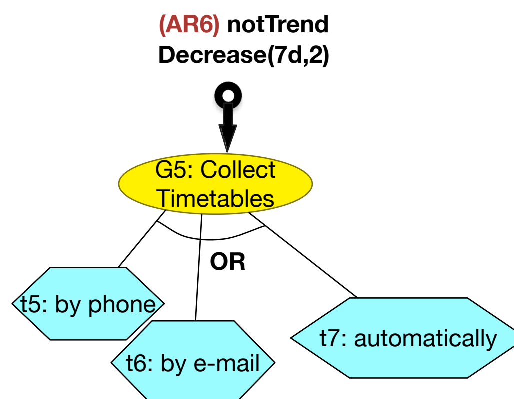


Figure 2.4: Trend Awareness Requirement.

A third type of *AwReq* are the Delta *AwReqs*. This type focuses on specifying acceptable thresholds for fulfilling the constrained goals. For instance, *AR*9 in Figure 2.5 specifies that when the meeting's date is scheduled manually, the task must be completed within one hour.

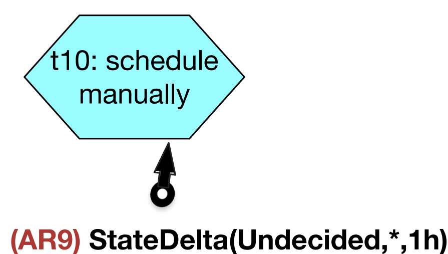


Figure 2.5: Delta Awareness Requirement.

AwReqs are implemented as constraints expressed in Object Constraint

Language (OCL) [Rob07] and the values of their success rates are captured by variables named indicators. For example, indicator $I1 = 85\%$ means that the 85% of the times the quality constraint for the goal Low Cost is satisfied and therefore, $AR1$ succeeds. Listing 2.1 depicts the OCL constraint for $AR1$ where $Q_CostLess500$ is the class of the quality attribute of the Low Cost soft goal.

```
-- AwReq AR1:  QC 'Weekly cost must be lesss than 500€' should have success
   rate 85%.
context Q_CostLess500
  def:  all: Set = Q_CostLess500.allInstances()
  def:  success : Set = all ->select(x | x.oclInState(Succeeded))
  inv AR1: always(success->size() / all->size() >= 0.85)
```

Listing 2.1: $AR1$ in OCL

2.1.4 Variability in goal models

For an adaptive system it is useful to implement all alternatives that are captured as OR-refinements in the goal model because this allows multiple reconfigurations during adaptation. Hence, some (in our example, all) OR refinements can be marked as *variation points* (see labels *VP1–VP3* in Figure 2.6). In this case, all tasks associated with each variation must be implemented and the system can switch from one configuration to another during adaptation [SLM11], as long as it adheres to its *behaviour model* (discussed next).

Another source of variability along the requirements dimension consists of *control variables*. These represent the amount of resources and effort allocated for the system-to-be while it fulfils its requirements. For instance, *FhM* represents *from how many* participants the system should collect time tables before goal *G5* is considered satisfied (a percentage value). *MCA* is another control variable that represents the *maximum conflicts allowed* for the timeslot chosen for the meeting and participant time tables. *RfM* is yet another, representing how many local (on the premises) rooms have been allocated for meetings, while, *HfM* represents how many hotel rooms

are reserved for meetings, and finally *VPA* indicates whether the system has authorization to access personal time tables.

Control variables and variation points, hereafter *requirement control parameters* (*ReqCPs*), can be adjusted at runtime by the adaptation mechanism, to fix failing *AwReqs*. The qualitative relation between *AwReqs* and parameters is captured through a systematic process called *qualitative system identification*[1]. During this process the domain expert captures the positive or negative influence that a parameter change can have on an *AwReq*. More specifically, the differential relationship $\Delta(I2/MCA) < 0$ means that by increasing *MCA* by one unit the success rate of *AR2* will decrease. Similarly, $\Delta(I5/MCA) > 0$ means that by increasing *MCA* the success rate of *AR5* will increase. Differential relations are symmetric with respect to increases/decreases, meaning that if *MCA* is decreased the success rate of *AR5* will also decrease.

2.1.5 Requirements Evolution

Requirements elicitation is not an easy task and stakeholders often change their minds during the development of the system-to-be or after the system is delivered. Moreover, setting thresholds for soft goals and constraints is not easy either, especially because some of these goals are conflicting and estimating an equilibrium is almost impossible until the system is deployed. In other cases, stakeholders have very high expectations from the system that cannot be always met due to external disturbances which are not captured during the design phase.

In order to cope with these challenges we use a new kind requirements, named *Evolution Requirements* (*EvoReqs*) [SLAM13]. This type of requirements specifies required changes to other requirements when certain

[1] In the original work presented in [SLM11] the process is referred simply as system identification. To avoid confusion with the system identification we present next for deriving analytical models, we add to this one the term qualitative

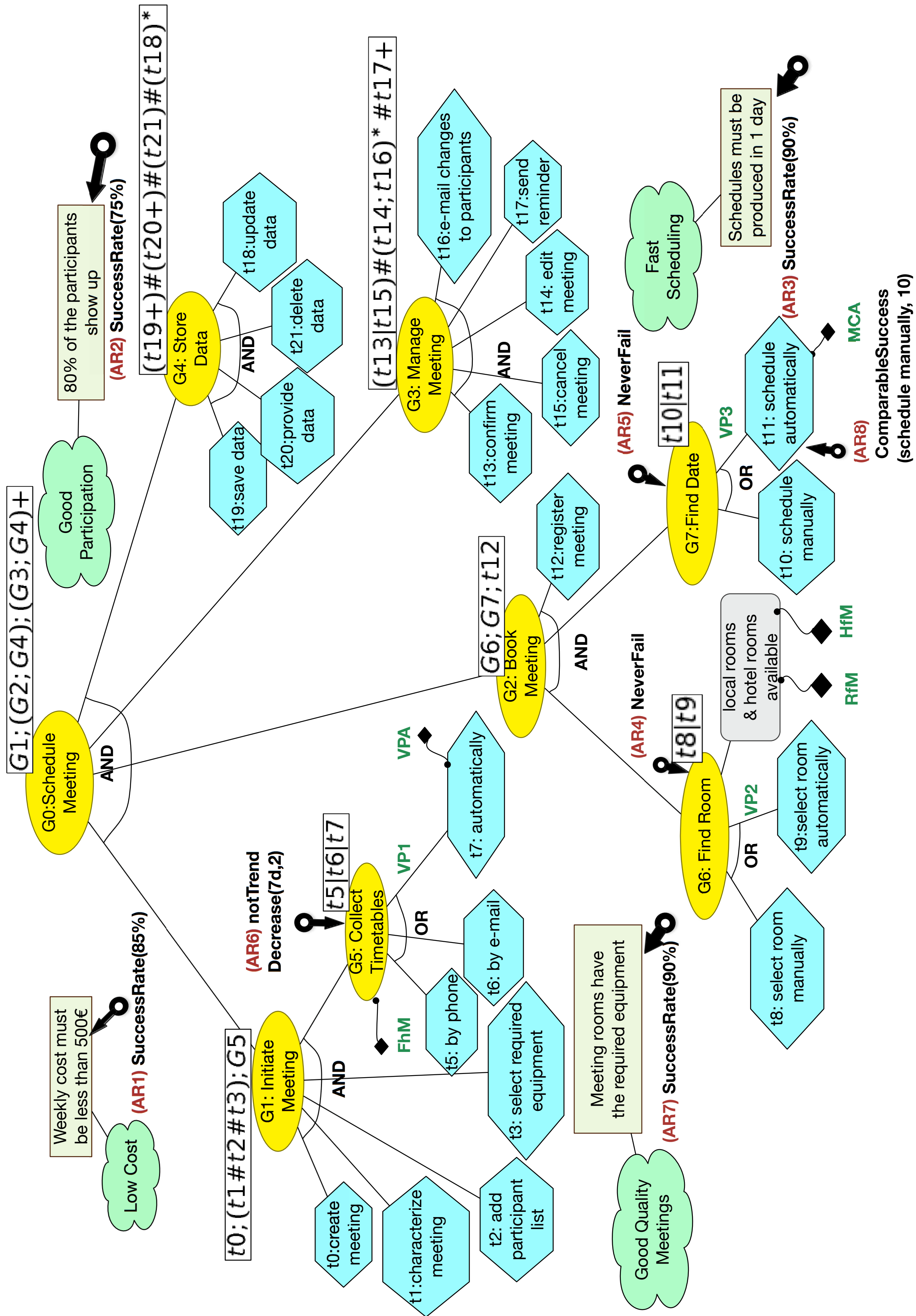


Figure 2.6: Goal model for the Meeting-Scheduler case study.

conditions apply (e.g., the failure of an *AwReq*). For example, If requirement R fails three times in a row, replace it with requirement R′, where R is a weaker (i.e., easier to fulfil) requirement. Such requirements are useful to evolve unfeasible requirements that were initially elicited from the stakeholders.

EvoReqs are applied by operations that are triggered by preconditions specified by stakeholders and designers. The triggering events can be a requirement failure of a scheduled event. Furthermore, the changes applied can either be permanent or temporary. Table 2.1 presents *EvoReqs* operations specified for the Meeting-Scheduler system. For example, when the system receives large amount of requests because a special event is taking place, or when the prices of the hotel rooms rise, the 85% threshold set by $AR1$ becomes infeasible. Therefore, when one of the aforementioned events takes place, the threshold is relaxed to 75% by the *EvoReq* operation Relax($AR1$,$AR1'$_75). When the environment returns to its previous state, meaning that the meeting requests are reduced, or the prices are decreased the threshold can be restored to the values by the *EvoReq* operation Strengthen($AR1$,$AR1'$_85). Other *EvoReq* operation might indicate that the system should wait for a certain amount of time before evaluating again the success of an *AwReq* as in the case of $AR6$ and $AR7$ or replace it permanently with another one such $AR5$.

2.1.6 Software Architecture Modelling

In [KOS06] Krutchen et al. describe software architecture as "*the structure and organization by which modern system components and subsystems interact to form systems, and the properties of systems that can best be designed and analyzed at the system level*". The software architecture is highly coupled to the requirements of a system since the latter prescribes what needs to be achieved and *why*, while the former describes *how* fulfil-

Table 2.1: *EvoReqs* operations

AwReq	*EvoReq* operation
$AR1$	1. Relax($AR1$,$AR1'$_75) 2. Strengthen($AR1$,$AR1'$_85)
$AR2$	Relax($AR2$,$AR2'$_90)
$AR3$	Relax($AR3$,$AR3'$_90)
$AR4$	1. wait(3 days) 2. Relax($AR4$,$AR4'$_75)
$AR5$	Replace($AR5$,$AR5'$_3)
$AR6$	wait(3 days)
$AR7$	wait(2 days)

ment is achieved.

More specifically, David Garlan in [Gar14] illustrates six aspects that software architecture contribute to software development. First, software architecture allows a better understanding of large and complex systems, since a high-level design is more comprehensible. Next, architecture design allows designers to reuse solutions to similar problems and facilitates the construction of the system-to-be. Moreover, when new components must be added or older ones are modified, the designers can reason about the impact on the system's integrity.

Architectures are described in terms of the concepts of components and connectors. A component constitutes the basic building block of architecture and is responsible for carrying out operations toward the fulfilment of goals. Components can be software, hardware components or human actors that interact with the system through an interface. Furthermore, components interact with each other within an architecture using communication links, named connectors. Multiple architecture description languages have been proposed during the years, such as ACME [GMW00], C2 [MTWJ96], Darwin [MDEK95], Koala [vOvdLKM00], Wright [AG94] and others. However to the moment this thesis is written, these languages

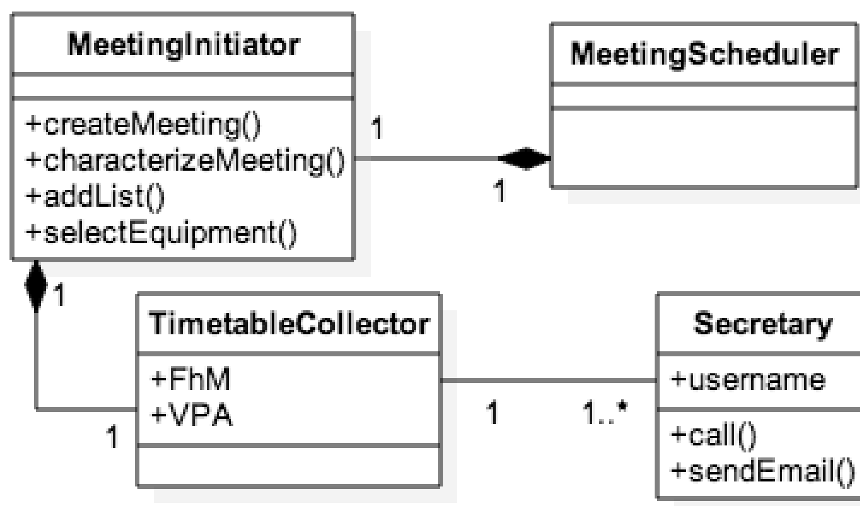


Figure 2.7: Architectural diagram for the Meeting-Scheduler

have been neglected by the software industry. Therefore, for our purposes, we use class diagrams to represent an architecture [ICG+04], where classes model components, while associations model connectors. Other types of relations between classes (e.g. composition) capture the structural relations of the system. For example, the class diagram in Figure 2.7 shows the architecture of the meeting scheduler system, where the TimetableCollector component is part of the MeetingInitiator component and can interact with one or more Secretary components.

Variability is captured in architectural models in terms of alternative components that can fulfil the same goal, but with different qualitative properties (e.g., better performance but lesser usability). Variability here can also be introduced by having a number of component instances participating in the runtime architecture. For instance, the meeting scheduler may have an additional component to what is shown on Figure 2.7 that takes in meeting scheduling requests and distributes them among one or more servers each of which consists of the architecture shown in Figure 2.7. This kind of variability is exploited by the Rainbow framework [GCH+04].

2.1.7 Software Behaviour Modelling

Another aspect of software systems is their behaviour and by that we mean the sequence in which the components execute their assigned tasks. There is rich literature on languages that describe software behaviour, such as Petrinets [Mur89], Statecharts [Har87], and BPMN [Whi04]. In this work, we represent the behaviour of the system using *flow expressions* [PCM+14, DBHM13] as attachments to each goal (in Figure 2.6, goals *G0–G7*). These are extended regular expressions that describe the flow of system behaviour, with each atomic component of allowed sequences of fulfilment of sub-goals that lead to the fulfilment of a parent goal.

The operators `;` (sequential), `|` (alternative), `opt()` (optional), `*` (zero or more), `+` (one or more), `#` (shuffle) allow us to specify sequences of system actions that constitute a valid behaviour. Shuffle specifies that its operands are to be fulfilled concurrently. For example, *G0 # G1* means that goals *G0* and *G1* are to be fulfilled in parallel. Of course, each of these goals has its own flow expression to describe in what order its own subgoals and tasks are to be fulfilled/executed.

Behavioural models contribute to disambiguating certain refinements of the goal models. For instance in Figure 2.6, Manage Meeting is AND-refined to five tasks. From a design point of view, this means that all the five functionalities must be supported by the system-to-be. However, at runtime, it is not an acceptable behaviour to cancel and confirm the same meeting.

2.2 Dynamic System Modelling

In the previous section we presented how the relations between control parameters and indicators can be captured using only qualitative information. Often, using qualitative adaptation is a necessity, given the lack of quanti-

tative models for software systems. However, in many cases, a sufficiently accurate analytical model, can be obtained through system identification techniques [Lju99], and can be used for control design. Letting $u(t) \in \mathbb{R}^m$ be the vector of m control parameter values at time t, and $y(t) \in \mathbb{R}^p$ be the vector of p indicators, their respective dynamic relation is described as:

$$y_i(t) = \sum_{j=1}^{p} \sum_{k=1}^{n_y} \alpha_{ijk} y_j(t-k) + \sum_{j=1}^{m} \sum_{k=1}^{n_u} \beta_{ijk} u_j(t-k) \tag{2.1}$$

for all $i = 1, \ldots, p$, and with $\alpha_{ijk} \in \mathbb{R}$, $\beta_{ijk} \in \mathbb{R}$. The quantitative dynamic model (2.1) relates the values of the indicator y_i at time t with past values of all the indicators – accounting for possible mutual influences of the indicators – and with past values of control parameters. For example, $I1$, that measures the success rated of the Low Cost goal in the Meeting-Scheduler exemplar, might achieve a high value because of good management of hotel room assignments or because of the constant failure of $I4$, the success rate of Find Room. The reason is that if meetings fail to be scheduled, no rooms are reserved and consequently the cost of meetings remains low. Such implicit relationships among indicators can be captured by model (2.1) to guide the adaptation process. Notice that if some of the mentioned variables are not influencing the value of the indicator $y_i(t)$, then the corresponding parameters are simply zero. An equivalent and more compact representation of this relation is the discrete-time state-space dynamic model:

$$\begin{cases} x(t+1) = Ax(t) + Bu(t) \\ y(t) = Cx(t), \end{cases} \tag{2.2}$$

where $x(\cdot)$ is a vector named *dynamic state* of the model. While for physical systems, the state $x(\cdot)$ is typically associated with meaningful physical quantities, in general the state can be just an abstract representation of

the system, and it is not necessarily measurable. The values of the matrices (A, B, C) fully describe how the inputs dynamically affect the outputs of the system, and these matrices are the outcome of the System Identification process.

The analytical model of Equation (2.1) shows that the system's output might be related to past outputs and control inputs. Indicators related to aggregate *AwReqs* [SLRM11] express success rates over time about the satisfaction of an associated goal and, therefore, their current values are naturally bound to their past values and to the values of *AwReqs* that produced them. These are *dynamic systems* in Control Theory. In case no relation with past behaviour of the indicators and of the control inputs is present, A is a matrix with all zero elements, and the system is just mapping inputs to outputs with the *static* relation:

$$y(t) = CBu(t-1).$$

Therefore, the model of Equation (2.2) accounts for both dynamic and static systems.

Equation (2.2) can be used to design a control system able to adjust the values of every control parameter, in order to make each indicator converge to the value prescribed by an *AwReq* threshold — under the assumption that the set of chosen control parameters is able to drive the system to the prescribed goals. In contrast to qualitative adaptation, such quantitative models allow one to handle conflicts with precision. For example, an increase of the control parameter MCA results in an increase of $I5$, the success rate of Find Date, as it becomes easier to find a commonly agreed timeslot for the meeting, but the participation might drop and consequently $I2$, the success rated of High Participation, is decreased. The analytical model can prevent the adaptation mechanism from decreasing $I2$ excessively. Performing such trade-offs on a daily basis while taking into

account priorities among indicators based on their business value (higher priority indicators should converge faster than less important ones), and preferences among control parameters (e.g. increasing RfM is preferred to increasing HfM) is a complex process. In chapters 7 and 8 we present a control-theoretic approach in order to efficiently implement this process and maintain an equilibrium among conflicting goals.

2.3 Related Work

In the past decade the literature in the area of self-adaptive systems has been enriched with multiple proposed frameworks, languages and techniques. In this section we present the state-of-the art of such approaches dividing them in four categories. First, we present approaches for software adaptation that use requirements, architectural and behavioural conceptual models or combinations of them. Finally, we conclude this Section with approaches that apply control theoretic techniques.

2.3.1 Requirements-based Adaptation

A well-known Requirements-based approach is RELAX [WSB+10], which aims at capturing uncertainty declaratively with modal, temporal and ordinal operators applied over SHALL statements (e.g., "the system SHALL ... AS CLOSE AS POSSIBLE to ..."). A recent extension of this work, AutoRELAX [FDC14], uses genetic algorithms in order to produce RELAXed goal models to change system requirements for avoiding failures caused by environmental uncertainty.

A similar approach, but based on the goal-oriented language KAOS [DvLF93], is FLAGS [BPS10]. This approach extends the linear temporal logic (LTL) used in KAOS with fuzzy relational and temporal operators, allowing some goals to be satisfied even if values are "around" but not ex-

actly equal to the desired ones. FLAGS also proposes an operationalization of its models in a service-oriented infrastructure.

The LoREM approach [GSB+08], also based on KAOS, uses an extension of LTL that includes an *Adapt* operator and defines a systematic process for performing goal-oriented RE for adaptive systems. Later, Cheng et al. [CSBW09] integrated this approach with the RELAX language in order to explore environmental uncertainty using threat modelling.

In [ZSL14] the authors propose the use of a cost-function for optimizing the non-functional requirements of the target system, captured by goal models, while minimizing the number the penalties taken for violating Service Level Objectives. The available adaptations are ranked with the use of the Analytic Hierarchy Process (AHP) [Saa80] by the designers before the system's deployment. Then, by applying a search-based method the adaptation which would optimize the cost-function is selected.

Another requirements-based and control theoretic approach is presented in [PCYZ10]. In this work the authors propose the use of a PID controller that finds a different configuration over a goal model that captures the system requirements. A SAT-solver is used to find the best configuration based on goal preferences. When soft-goals are not met, the controller tunes the values of the assigned preferences in order fors the SAT-solver to find a better configuration.

There are also a few RE-based approaches for the design of adaptive systems based on i^* [YGMM11] and Tropos [BPG+04]. Tropos4AS [MPP09] is a methodology for the design of agent-based adaptive systems founded on the Belief-Desire-Intention (BDI) model. As run-time infrastructure, Tropos4AS proposes the mapping of goal models to Jadex.[2] The CARE method [QP10] also bases itself on Tropos, but focuses on service-based applications. Adaptive requirements are specified at design time and a

[2] A BDI Agent System, see `http://jadex-agents.informatik.uni-hamburg.de/`.

run-time infrastructure based on environment monitoring, service selection and customization is provided. Dalpiaz et al. [DGM12] propose an architecture that adds self-reconfiguring capabilities to a system using a Monitor-Diagnose-Compensate (MDC) loop based on the system's requirements models in i^*. Different reconfiguration algorithms are proposed on top of this architecture.

2.3.2 Architecture-based Adaptation

In [OGT+99] Oreizy et al. propose one of the first reference frameworks for architecture-based adaptation. On the foundations of this approach there is an architectural model constructed using C2 [MTWJ96] which captures the properties and the component structure of the system. The purpose of the architectural model is twofold. First, it allows to maintain the integrity of the system as the system evolves by adding and removing components. Next, the proposed reference framework includes a planning mechanism that produces adaptation plans on the fly to cope with the environment's uncertainty, The architecture model facilitates the planning process by giving an overview of the systems status and hence locating where changes are required.

On the side of architecture-based approaches, Rainbow [CGS06b] is a well known framework that we have also used to represent architecture-based adaptation approaches to perform a comparative study, illustrated in the next chapter. The adaptation space of Rainbow is captured by ACME architecture models and the control parameters are instances of components or properties of the latter. The adaptation strategies are meant to automate administration tasks performed by humans and are captured by a script language named Stitch [CG12]. Then the framework that is an implementation of the MAPE loop we presented earlier, uses Utility Theory in order to select which adaptation strategy is most suitable for every

failure. In the same research line, in [CGSP15] the authors propose the use of probabilistic model checking techniques to compose dynamically adaptation strategies taking also into account latencies about when the impact of a change in a control parameter will appear to the system's output.

Sykes et al. in their work [SHMK10] assign utility properties to all components of the system. Dependency graphs are used to capture component constraints and each component is annotated with utilities that indicate how it will improve or harm the non-functional requirements of the system. When one or more requirements fail the proposed adaptation mechanism finds a new component composition that maximizes the the overall utility and at the same time respects the architectural dependencies and constraints. Similar to this approach, the SASSY framework uses optimization as a decision-making mechanism to decide alternative service compositions for Service Oriented Architectures (SOA) [MGMS11]. In the same context Foster et al. propose an online reconfiguration process for SOA, that exploits prediction models in order to anticipate environmental changes such as workload peaks.

Another architecture-based approach is presented in [SBP+08]. In this work, the authors propose a resource provisioning system that allows the users to state their preferences about the quality of service of the system. For instance, the users have to choose if latency or accuracy is more important for them and as well as they expected thresholds they expect the system to comply with. Therefore, the adaptation framework will perform trade-offs based on the user preferences producing adaptation plans that include resource provisioning and forecasting.

Flashmob [SMK11] is yet another architecture-based approach for engineering self-adaptive systems. The main focus of Flashmob is the component distribution and how loosely components can coordinate in order to cope with failing nodes of the system. Flashmob exploits a communica-

tion protocol gossip which is used by the system's components in order to inform each other about their status. When a requirement fails or a component is malfunctioning, the healthy components which are individually aware of the system's architecture and goals, coordinate in order to deploy new components or assign tasks to existing ones in order for the system to recover.

2.3.3 Behaviour-based Adaptation

In [LYM07], Lapouchnian et al. describe how to derive high variability business process models that capture the system's behaviour from goal models. The behaviour derivation process is carried out by annotating flow expressions to goal models as it is demonstrated in Section 2.1.7. Then, these expressions are converted to BPEL [Jur06] processes. Finally, using the contributions of the hard goals to the soft goals specified by the stakeholders and the system designers a configuration is decided. The contribution of this proposal is the construction of high variability business processes that can adapt to changes of stakeholder preferences.

Another behaviour-based approach is the CEVICHE framework [HSD10] which uses a Complex Event Processing engine to identify exception to the regular business process. These exceptions are handled with predefined adaptation that are encoded in a BPEL variation, namely SBPEL. This case-based adaptation mechanism allows to maintain a specific level of Quality of Service (QoS) without having to implement all the potential variations of the business process since the system can reconfigure dynamically.

In the same line of research with the previous approaches VxBPEL [KSSA09] is one more variation of BPEL for adaptation purposes. VxBPEL allows the user to define high variability workflows. Then, the adaptation engine is capable of reconfiguring the business process on-the-fly, by select-

ing values for the variation points of the workflow, based on the input of the adaptation process.

2.3.4 Combined Model-based Adaptation

Having as a starting point a goal model, Yu et al. [YLL+08] propose heuristics to derive other models such as feature models, statecharts and component-connector models. Their purpose is to express the same level of variability in different dimensions of the system.

The STREAM-A approach presented in [PLC+12] derives ACME architectural models from goal models using model transformations. The environment's influence on the requirements is captured in terms of context. The main purpose of this work is to relate the requirements to components and place accordingly the actuators and the sensors of the adaptation mechanism.

In [SHMK08] goals and components are related with reactive plans. When a failure takes place or a goal is changing, the proposed adaptation mechanism generates a new plan of actions that needs to be carried out and the available components that are required are reconfigured to the current architecture. This approach demonstrates the advantages of architectural variability, by assigning goals to multiple components.

Chen et al. in [CPY+14] propose the combination of goal models and architectural decisions in order compose a larger adaptation space. More specifically, the tasks of the goal model are assigned to one or more components that can be used interchangeably. Then, the adaptation mechanism can select alternatives between the variation points of the goal model and the alternative components available.

2.3.5 Control-based Adaptation

Most approaches that have been proposed have in common the adoption of the concept of feedback loop from Control Theory. As we mentioned earlier Control Theory has solved in multiple domains of other engineering disciples adaptation problems, where a quantitative goal and one or more control parameters are available. Recently, there has been some significant effort on introducing control engineering approaches in the development process of self-adaptive software systems [FMA+15]. Hereby, we present approaches that have applied formal control theoretic techniques in order to develop adaptive software systems.

One of the first proposals that builds a controller as an adaptation mechanism is presented in [PGH+01]. In this work the authors use an analytical model to capture how the allowed number of remote procedure calls to an IBM Lotus Domino server affect its response time. Therefore, the analytical model captures the relationship of these two variables through time. Then this information is used in order to build an integral controller that can stabilize the response time to the given reference input. In addition to this work, building various kinds for controlling computing systems [HDPT04] and resources in operating systems [LMPT13] are present in literature.

An automated solution to introduce control in a seamless way was proposed in [FHM14]. This solution treats Single Input and Single Output (SISO) systems by varying a single input and measuring the output. The solution builds on a simple and qualitative dynamic model which is identified online. More precise yet complicated models can be used at the cost of a higher overhead at runtime [ABG+13]. However, this approach works only for SISO systems, while the case of Multiple Inputs and Multiple Outputs (MIMO) cannot be addressed. In the same line of research the authors

extended their approach in order to deal with MIMO systems [FHM15] where the MIMO control is obtained as an automated synthesis by composing SISO controllers in a hierarchical way.

Finally, in the domain of Cloud Computing variations of Model Predictive Control (MPC), which we discuss in detail in chapter 7 have been applied extensively. In [GC15, KKH+09] the authors apply look-ahead control to improve the energy consumption and the performance of the cloud. Similarly, in [GLPB14] MPC is applied to improve the replica placement mechanism and deal with multiple SLOs.

All these approaches offer significant improvements to their respective applications, although are highly customized to the specific problem they are solving. On the other hand, our approach is more generic and therefore easier for software engineers that have no expertise on Control Theory to use it. Moreover, in our work we integrate control design and requirements engineering in order to provide a guideline about to how to integrate MPC with the development of self-adaptive software.

2.4 Chapter Summary

In this chapter, we summarized the baseline for presenting our proposals through this thesis. This work is a continuation of a requirements-based adaptation approach presented in [SS12] and therefore, most of its components constitute the foundations of our research. First, we presented the concept of *AwReqs* in order to monitor the success of other requirements. Next, we explored model for capturing three important dimensions of software systems, requirements, behaviour and architecture, with main focus on variability. We presented a qualitative system identification process to model the relationships between the system's variables and the success of its requirements.We also discussed another kind of requirements, namely

EvoReqs that prescribe how other requirements should change over time. The last piece of our baseline describes how a system can be described using analytic models and how the latter are related to the system's goals and variables.

Finally, this chapter summarizes different approaches for implementing adaptation mechanisms. We mainly categorized these proposal by the kind of models they are using in each of the three dimensions of software systems we investigate in this thesis.

Chapter 3

Requirements and Architecture Approaches: A Comparison

> Computer Science is a science of abstraction - creating the right model for a problem and devising the appropriate mechanizable techniques to solve it.
>
> A. Aho and J. Ullman

In Section 2.3 we presented various proposals, intended to guide developers in the development of self-adaptive systems, some focus on architecture models that capture architectural variability and support reconfigurations in the system's structure, propagating the effects to the actual system, in response to certain situations. Instead, other approaches, advocate the use of requirements models to capture variability and support adaptation.

This dichotomy has motivated us to investigate whether these two types of approaches can produce the same results, what are their respective advantages and drawbacks, and study whether they are complementary, providing answers to **RQ1**.

In this Chapter we present a comparative study of one representative approach of each of the aforementioned categories, respectively: Rainbow [GCH+04] and *Zanshin* [SS12]. Our methodology consisted of applying

both frameworks to the same exemplar: the ZNN.com case study presented in [CGS09] for the Rainbow framework. Models of the system's adaptation rules were produced for each framework and adaptation scenarios based on an implementation of *ZNN.com* were executed.

3.1 Selected Adaptation Approaches

In the previous chapter we overviewed several approaches that use various kinds of models in order to adapt when they fail to fulfil their mandate. In the literature the two most common models used for software adaptation capture either the requirements or the architecture of the target system. Therefore, we selected a representative framework for each category to analyze the characteristics of both kinds of models and their role in the adaptation process.

- ***Requirements-based*** (henceforth ***RE-based***) approaches: extend Requirements Engineering techniques in order to represent the requirements of adaptation and/or the inherent uncertainty of the environment in which the system operates. These approaches may or may not include mechanisms for runtime reasoning and frameworks that operationalize the adaptation requirements, since they focus on capturing and analyzing the problem rather than implementing solutions.

- ***Architecture-based*** approaches: concentrate on helping designers build architectures that support adaptation. They usually propose the use of an architectural model that shows system components and how they communicate amongst themselves through connectors. Such proposals often include the runtime software infrastructure on top of which to build the adaptive system, taking care of its adaptation rules and how to evolve its models.

As we mentioned earlier, for our comparative study, we selected the *Zanshin* and *Rainbow* to represent requirement and architecture-based adaptation frameworks respectively. These frameworks were chosen for a number of reasons. Firstly, they are good representatives of their respective schools of thought on building adaptive software systems. Secondly, they are fairly comprehensive and quite well documented in guiding the design of adaptive systems. Thirdly, there was code readily available for running our experiments. We summarize both approaches next.

3.1.1 Rainbow

The *Rainbow* framework [GCH+04] is a prominent architecture-based approach for the design of self-adaptive systems. According to the proposal, adaptation rules are used to monitor the operational conditions of the system and define actions to be taken if the conditions are unfavourable. For example, given a news website (which we will detail in Section 3.2), if measured response times are too long, actions such as enlisting more servers or switching from multimedia to textual mode can be executed to try and improve response time.

The framework prescribes the use of the ACME architecture description language [GMW10], which extends the usual component-connector representation with the concept of *families*, allowing designers to define different architectural variants and styles [SG02]. This allows for the specialization of the framework to specific application domains, defining style-specific architectural operators and repair strategies [GCS03].

Figure 3.1, adopted from [Che08], shows the elements that compose the *Rainbow* framework. Monitoring is done with a set of *Probes* deployed in the target system, which send observations to *Gauges* that interpret the probe measurements in terms of higher-level models. The *Model Manager* is responsible for tracking the changes in the models' states and keeping it

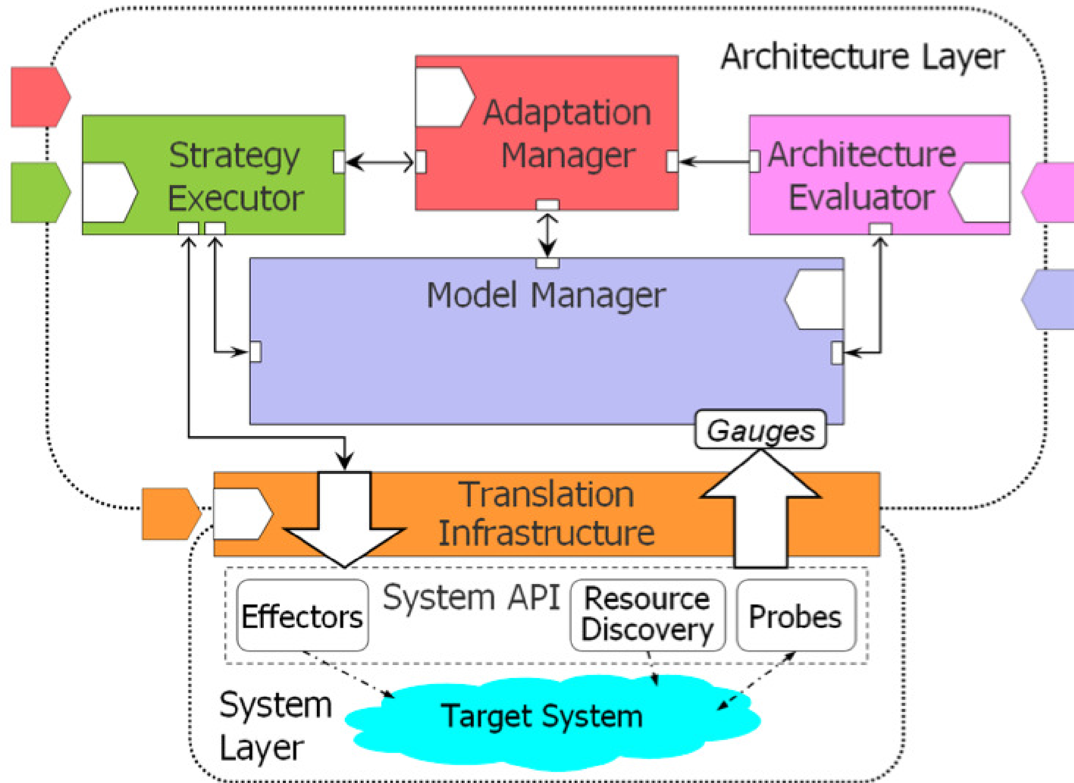


Figure 3.1: The components of the *Rainbow* framework [Che08].

consistent with the target system. Moreover, other components query the Model Manager for information about the current state of the model.

One of these components is the *Architecture Evaluator*, which detects changes in the status of the properties of the system's architecture and environment, validating such changes with respect to the constraints stated in the model. In case of a violation, it triggers the *Adaptation Manager* in order for it to select the most appropriate strategy, using Utility Theory (details in [Che08]) for the decision. Finally, the *Strategy Executor* coordinates the execution process, deciding the operators that should be applied through the *Effectors* at the *System Layer*.

For the final parts of the adaptation loop, *Rainbow* uses a language called *Stitch*, which captures routine human adaptation knowledge as explicit adaptation policies [CG12]. The language allows designers to specify

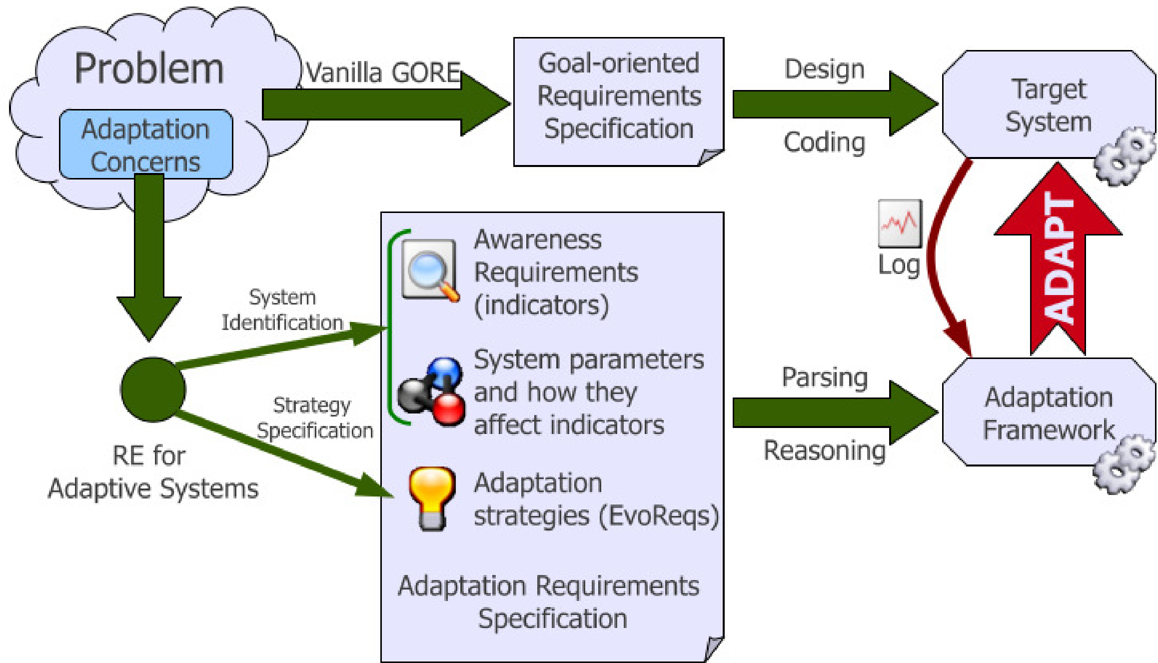


Figure 3.2: An overview of the *Zanshin* approach [SS12].

what, when and how to adapt, thus automating the adaptation process. In Section 3.2 we will see some examples of Stitch applied to the exemplar chosen for our experiments, the news website *ZNN.com* [Che08].

3.1.2 *Zanshin*

Zanshin, is an RE-based framework for the design of adaptive systems that exploits concepts presented in the previous section such as *AwReqs*, *EvoReqs* and feedback loops to design adaptive software systems [SS12]. The core idea of the approach is to make the elements of the feedback loops that provide adaptivity first class citizens in the requirements models. An overview of the approach is shown in Figure 3.2. In particular, *Zanshin* uses *AwReqs* is its monitoring mechanism and differential relations as a basis of its adaptation.

Strategy Specification focuses on the adaptation part of the feedback

loop. Its objective is to associate one or more *adaptation strategies* (e.g., "Retry/delegate a task", "Relax a requirement", etc.) with each *AwReq* in order to have them executed in case of an *AwReq* failure at runtime. These strategies should also be elicited from the stakeholders and are represented by *EvoReqs*. As we explained in Section 2.1.3, *EvoReqs* prescribe how other requirements of the model should evolve in response to an *AwReq* failure, and are specified using a set of primitive operations, each of which is associated with application-specific actions to be implemented in the system. One strategy in particular, the *Reconfiguration* strategy, uses the information elicited during qualitative System Identification to reconfigure the system, also allowing designers to specify different reconfiguration algorithms depending on the amount of information available.

A prototype framework that operationalizes a feedback loop based on the models produced by *Zanshin* is available at `https://github.com/sefms-disi-unitn/Zanshin`. The experiments described in this chapter (cf. Section 3.3) were conducted using this framework and can be repeated by the interested reader. In the next section, we will derive a goal model to represent the requirements of the *ZNN.com* exemplar used in the experiments and apply *Zanshin* to it.

3.2 The *ZNN.com* Exemplar

An exemplar, or a *model problem*, is a shared, well-defined problem adopted by researchers of a specific field for presenting and comparing proposals. The *Software Engineering for Adaptive and Self-Managing Systems* (SEAMS) research community has proposed some exemplars in their website,[1] among which we chose *ZNN.com* to perform the comparative study presented in this chapter.

[1]See `https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/`.

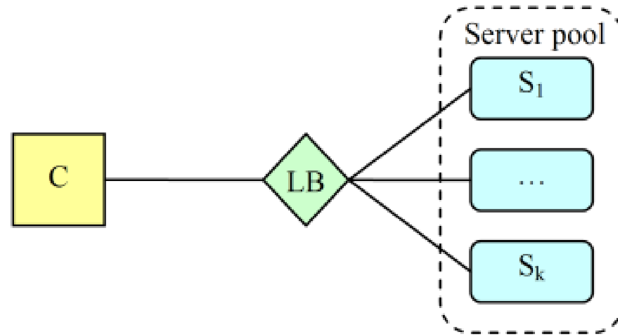


Figure 3.3: Znn.com architecture [CGS06b]

The choice of *ZNN.com* was also motivated by the fact that it had already been used in [Che08] as a case study for the proposal of the *Rainbow* framework. In this section, we present an overview of this model problem and how it was solved by *Rainbow*; then we apply *Zanshin* to it in order to be able to compare these two approaches.

3.2.1 Overview of the problem and its architectural solution

ZNN.com is a news service that serves multimedia news content to its customers through a website. It is a simplified version of real sites such as `cnn.com`.

ZNN.com's adaptive features are needed when the website experiences spikes in news requests due to, for instance, popular events. In these cases, response times for user requests might become unacceptable and the system has two possible adaptation strategies: enlisting new servers to divide the load of requests or switch from multimedia to text-mode to make each request quicker to respond. However, these strategies may cause problems in two other requirements of this system: first, the website managers would like to run the system at the lowest cost possible and adding new servers costs money; second, the users would like to see news with high content fidelity (i.e., high presentation quality), preferring multimedia over simple

text.

Like most high demand websites, the architecture for *ZNN.com*, depicted in Figure 3.3 includes a load balancer (LB) that distributes requests among multiple web servers and a database server. Current technology for load balancing and cloud computing already supports some level of adaptation, but automating trade-offs among multiple objectives like stated above is usually not supported [Che08]. For the *ZNN.com* case study, the operational target of the system is to keep a balance among its cost, performance and content fidelity.

The challenge of such systems is to achieve their mandate even when they operate under critical conditions. The difficulty lies in taking the right decision at the right time, in the sense that the problem should be detected promptly and the most efficient strategy to stabilize operation should be applied immediately. Under such circumstances, human intervention can be insufficient and automated mechanisms are required to carry out both decision making and adaptation.

Rainbow tackles this challenge through a software architecture model of *ZNN.com* written in the ACME language. The model includes elements representing *clients*, *servers*, *connections* and the *proxy*; as well as properties for the client's *experienced response time*, the connection's *bandwidth* and the server's *cost*, *load* and *fidelity*. Moreover, operations for (de)activating a server and setting its fidelity allowed architectural designers to create four tactics that can be applied when adaptation is necessary: enlisting/discharging servers and raising/lowering the fidelity [CGS09].

Tactics such as these are combined in strategies, written in Stitch to form high level adaptation processes. The exact definition of the adaptation strategies used in *ZNN.com* are described in Appendix C of Shang-Wen Cheng's thesis [Che08]. We show one of these strategies in Figure 3.4 and summarize them below:

```
strategy SmarterReduceResponseTime
[ styleApplies && cViolation ] {
  define boolean unhappy = numUnhappyFloat/numClients > M.TOLERABLE_PERCENT_UNHAPPY;

  t0: (unhappy) -> enlistServers(1) @[500 /*ms*/] {
    t1: (!cViolation) -> done;
    t2: (unhappy) -> enlistServers(1) @[2000 /*ms*/] {
      t2a: (!cViolation) -> done;
      t2b: (unhappy) -> lowerFidelity(2, 100) @[2000 /*ms*/] {
        t2b1: (!cViolation) -> done;
        t2b2: (unhappy) -> do[1] t2;
        t2b3: (default) -> TNULL;  // in this case, we have no more steps to take
      }
    }
  }
}
```

Figure 3.4: Strategy `SmarterReduceResponseTime` in Stitch [Che08].

- `SimpleReduceResponseTime`: In case a client experiences response time above a predefined threshold then the fidelity is lowered by one step. In case response time remains high, fidelity is decreased again one more step;
- `SmarterReduceResponseTime`: If an unacceptable percentage of clients experiences high response time, then enlist one server, then enlist another server and finally lower fidelity by one step. Repeat twice the last two actions until response time is restored;
- `ReduceOverallCost`: If server cost is higher than a threshold value then reduce the number of servers by one. If response time is low and cost remains high repeat the previous action, until cost is returned to normal;
- `ImproveOverallFidelity`: If content fidelity level is below threshold then raise fidelity of all servers by one step. If response time is low and fidelity remains low then raise fidelity level one more step.

The strategies above compose the possible options of the *Adaptation Manager* we described earlier when it is required to restore the system's

invariants such as cost, performance and content fidelity to their desired levels.

Given the availability of *Rainbow* models for the *ZNN.com* exemplar, to conduct the comparative study we needed to produce models of the system according to *Zanshin*. The results of this effort are reported next.

3.2.2 An RE-based solution to *ZNN.com* using *Zanshin*

Using available documentation, we have elicited requirements for the *ZNN.com* exemplar, producing the model shown in Figure 3.5. Of course, the figure does not represent complete requirements for a news service (which would include concerns such as adding news, searching, managing advertisement, etc.), but concentrates on the adaptation scenario described earlier.

Requirements for the system are represented using Goal-Oriented Requirements Engineering (GORE) elements such as *goals*, *tasks*, *softgoals*, *quality constraints* and *refinement* relations that indicate how (soft)goals are satisfied using Boolean semantics as we discussed in Section 2.1. One of *ZNN.com*'s goals is to *Serve news* to its visitors, which can be accomplished using *text-only*, *low resolution* or *high resolution* contents. Three non-functional requirements also compose this simple scenario:

- *Cost-efficiency*: the system should either be operating using a single server, unless response times are above a certain minimum threshold (MIN_{RT}), which would justify the addition of extra servers;
- *High fidelity*: analogously, the system should prefer high resolution content over lower ones, unless response times are above the minimum threshold;
- *High performance*: response time and server load should be under a certain maximum threshold (MAX_{RT}).

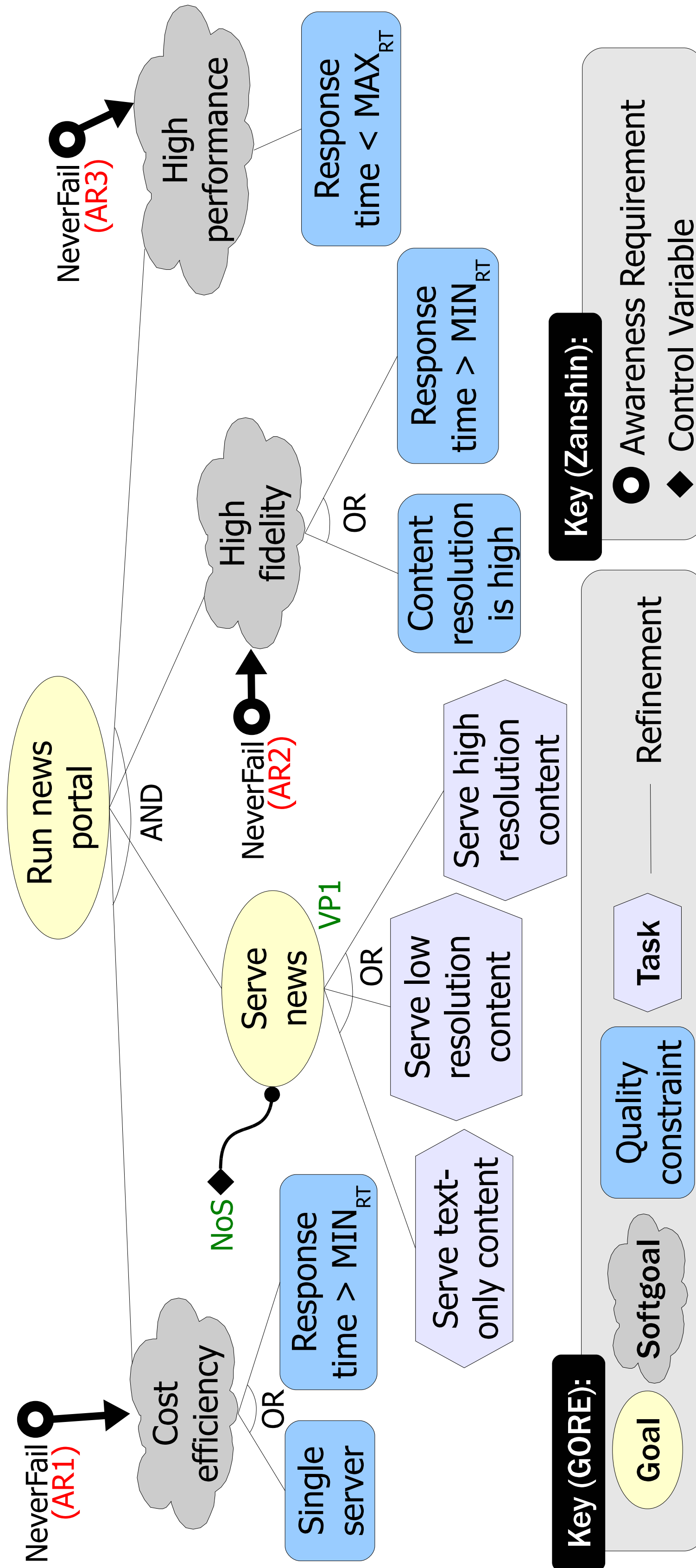


Figure 3.5: Goal model for the *ZNN.com* exemplar, mirroring the adaptation scenarios modelled in *Rainbow*.

Quantitative values for MIN_{RT} and MAX_{RT} should eventually be provided by the stakeholders, but are not essential to the discussion herein.

On top of this base model, we have applied *Zanshin* to elicit requirements for adaptation as well. In the model of Figure 3.5, these are represented by *AwReqs AR1*, *AR2* and *AR3*, variation point *VP1* and control variable *NoS* (*Number of Servers*). Moreover, by applying System Identification to *ZNN.com* we have come up with the following qualitative relations among indicators and control parameters:

$$\Delta\,(I1/NoS)\,[0, maxServers] < 0 \tag{3.1}$$

$$\Delta\,(I3/NoS)\,[0, maxServers] > 0 \tag{3.2}$$

The equations tell us that increasing the number of servers will hurt cost-efficiency (3.1) decreasing the indicator $I1$, but contribute toward higher performance (3.2) by increasing the indicator $I3$. Relations between variation point *VP1* and *AwReqs AR2* and *AR3* were also identified, but are not necessary to produce the *ZNN.com* scenarios presented in the previous subsection (we come back to those in Section 3.3.3). Finally, Figure 3.6 shows the complete specification for *AwReqs AR1–AR3*, based on *Rainbow*'s `SimpleReduceResponseTime` strategy presented earlier.

Assuming initial values $NoS = 4$ and $VP1 = \textit{high resolution}$, *AR1* will never actually fail (the simple scenario does not include enlisting of servers) and *AR2* (checked for every user request) will not fail initially. When *ZNN.com* experiences spikes in news requests it may cause *AR3* (related to *High performance* and also checked at every request) to fail, starting an adaptation session for it.

The first adaptation strategy (AS3.1), applicable only for the first failure of the session, is to change parameter *VP1*, which will take the value *low*. For the next $1000ms$, other requests with response time over threshold

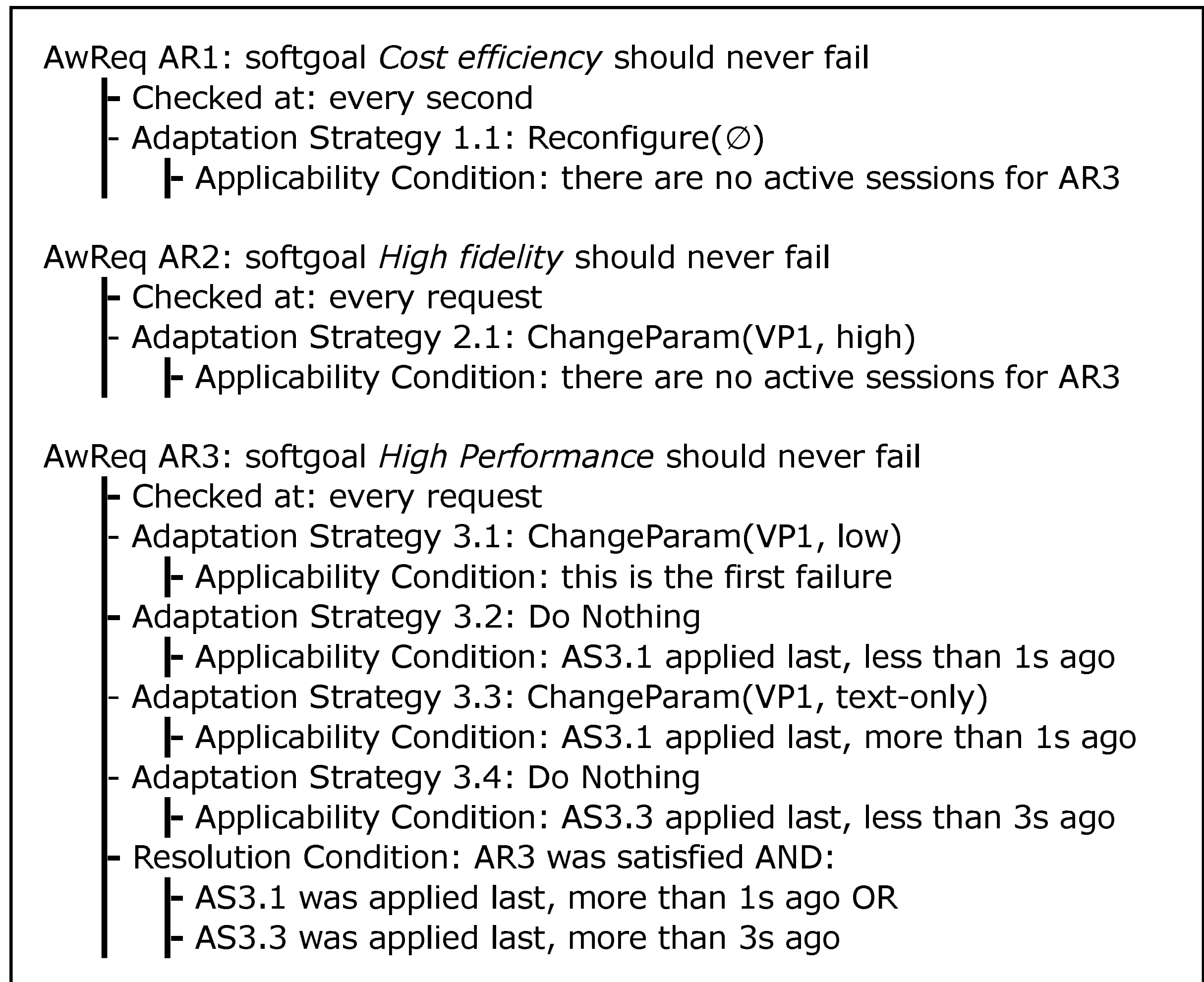


Figure 3.6: Specification of the `SimpleReduceResponseTime` strategy with *Zanshin*.

will activate strategy 3.2 (*Do Nothing*), simulating the waiting period of *Rainbow*'s strategy.

If *AR3* keeps failing for more than a second, AS3.2 will cease to be applicable, giving turn to AS3.3, which switches the fidelity to text-only mode and becomes immediately inapplicable. Further failures in the next 3 seconds will activate AS3.4 to simulate another waiting period. If the problem is not solved during this entire time, the *Abort* strategy (by default, the last resort in all *AwReq* failures) will take place and close the session.

The problem is considered solved when its resolution condition becomes true. This means that *AR3* was evaluated as being satisfied for a request that happened after one of the useful strategies (i.e., the ones that are not *Do Nothing*) was applied. Until this is true and *AR3*'s session is closed, failures of *AR2* will not lead to its strategy being executed, given its applicability condition. Once *AR3* is done and response time goes under MIN_{RT}, *AR2*'s strategy will be applicable and, as a result of its execution, *ZNN.com* will go back to serving multimedia content.

As the description above shows, the models produced by applying *Zanshin* can produce, at runtime, the same result as the *Rainbow*'s `SimpleReduceResponseTime` strategy. It is also possible to model the `SmarterReduceResponseTime` strategy, by changing the specification of *AR3*, as follows.

First, its definition would change from "*High performance* should never fail" to "*High performance* should not fail for more than the $MAX_{unhappy}$% of the clients", where $MAX_{unhappy}$ represents the percentage of clients who experience high response times that is tolerated before something has to be done. Second, its adaptation strategies and resolution condition should also change, as shown in Figure 3.7.

The new specification of *AR3* represents, in a declarative way, the same algorithm described for `SmarterReduceResponseTime` in the previous subsection: reconfiguration (enlisting of additional servers) is applied at first with half a second of wait, then multiple times more (as long as the number of servers does not overcome $maxServers$) interposed with gradual reductions of fidelity.

The above exercise of mapping *ZNN.com*'s *Rainbow* specification to *Zanshin* indicates that the latter, although using a different representation, has at least equivalent expressiveness to the former. We will come back to this in the discussion of Section 3.3.3.

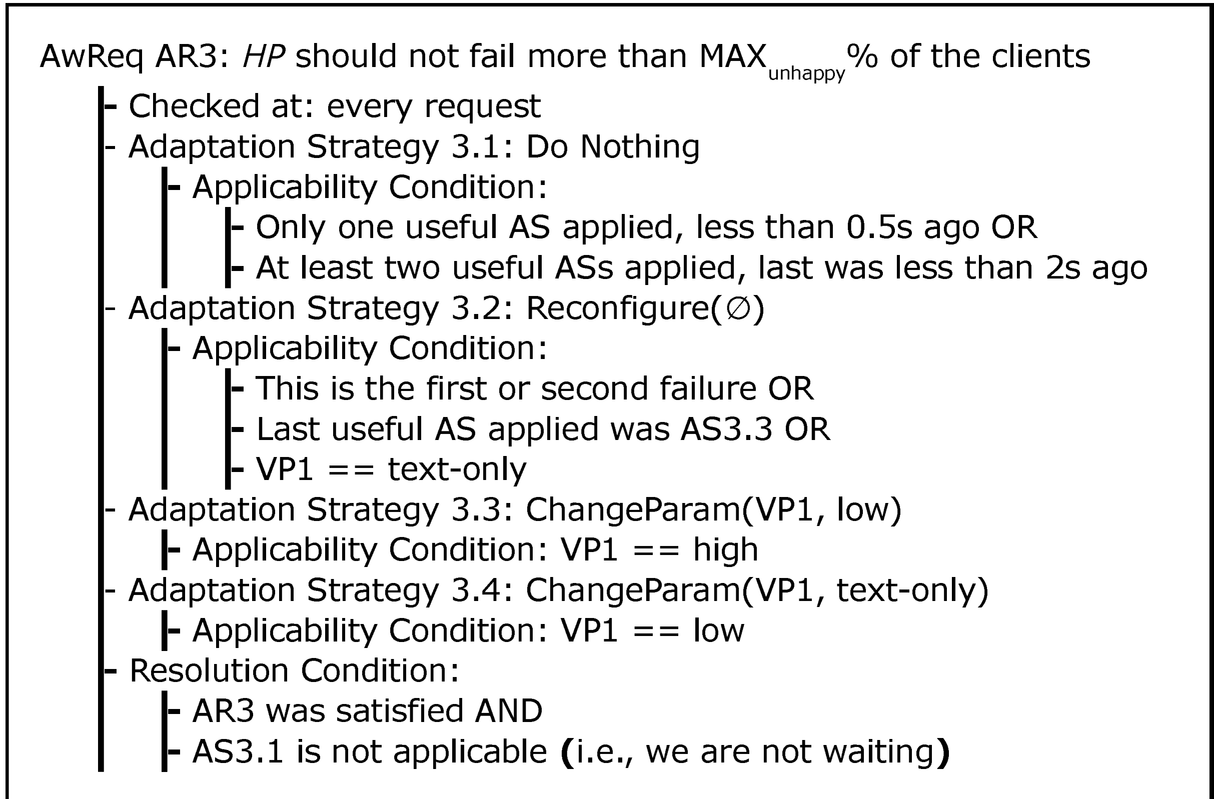


Figure 3.7: Specification of *AR3* for the `SmarterReduceResponseTime` strategy.

The expressiveness of *Zanshin* is due to its extensibility, allowing for new *AwReq*/*EvoReq* patterns and applicability/resolution conditions to be created. In effect, most of the conditions used in the examples of Figure 3.6–3.7 did not yet exist when we started this experiment.

3.3 Comparison between Rainbow and Zanshin

In the previous section we represented the adaptation strategies that *Rainbow* implements using Stitch with the *EvoReqs* of *Zanshin*. This allowed us to repeat the same experiment that simulates a scenario of highly increasing traffic that was already implemented for *Rainbow* [CGS09], but this time assigning the adaptation control to *Zanshin*.

It is important to point out that the purpose of this work is not to

compare the performance of the two frameworks or provide a better solution for the case study, but to compare and contrast the two approaches. To this end we mirror the solution of the *ZNN.com* case study using the *Zanshin* framework in order to make an apples-to-apples comparison. In what follows, we describe the methodology of our experiment, present its result and discuss the two frameworks based on our experience.

3.3.1 Methodology

For purposes of this work, we implemented in *Zanshin* the goal model shown in Figure 3.5, along with the specification of the `SimpleReduceResponseTime` strategy described in Figure 3.6. For the base system (the *ZNN.com* website) we used the source code available on the SEAMS community website.

The deployment configuration is similar to the one described in [CGS09] for the evaluation of the *Rainbow* framework and includes five Apache web servers (four replicated hosts and one proxy) and a MySql database server running on a Debian-flavored operating system. A Java™ application called JMeter, which is used to perform stress tests on web applications, is instantiated in one additional machine that plays the role of the clients who send requests to the server. The workload we created for the experiment (equivalent to that of [CGS09]) simulates a real world case that many websites like *ZNN.com* deal with on a regular basis. The traffic scenario is as follows:

1. Slow start with 6 visits/min;
2. Sudden increase for five minutes where the traffic increases by 120 visits/min every minute until it reaches 600 visits/min;
3. Hold the load for 18 minutes;

4. For the remaining 36 minutes reduce the workload by 15 visits/min every minute.

After running the experiment and evaluating the effectiveness of *Zanshin*, we compare the characteristics of the two approaches by indicating weaknesses and advantages. The comparison points we set include a) adaptation type b) the kind of models used by each approach, c) the adaptation actions, d) the adaptation triggering, e) the adaptation selection and f) how each framework deals with adaptation failures. The outcome of this comparison can be exploited by the ongoing research on adaptive systems, leading to adaptation frameworks that would combine the maximum set of advantages of the current approaches.

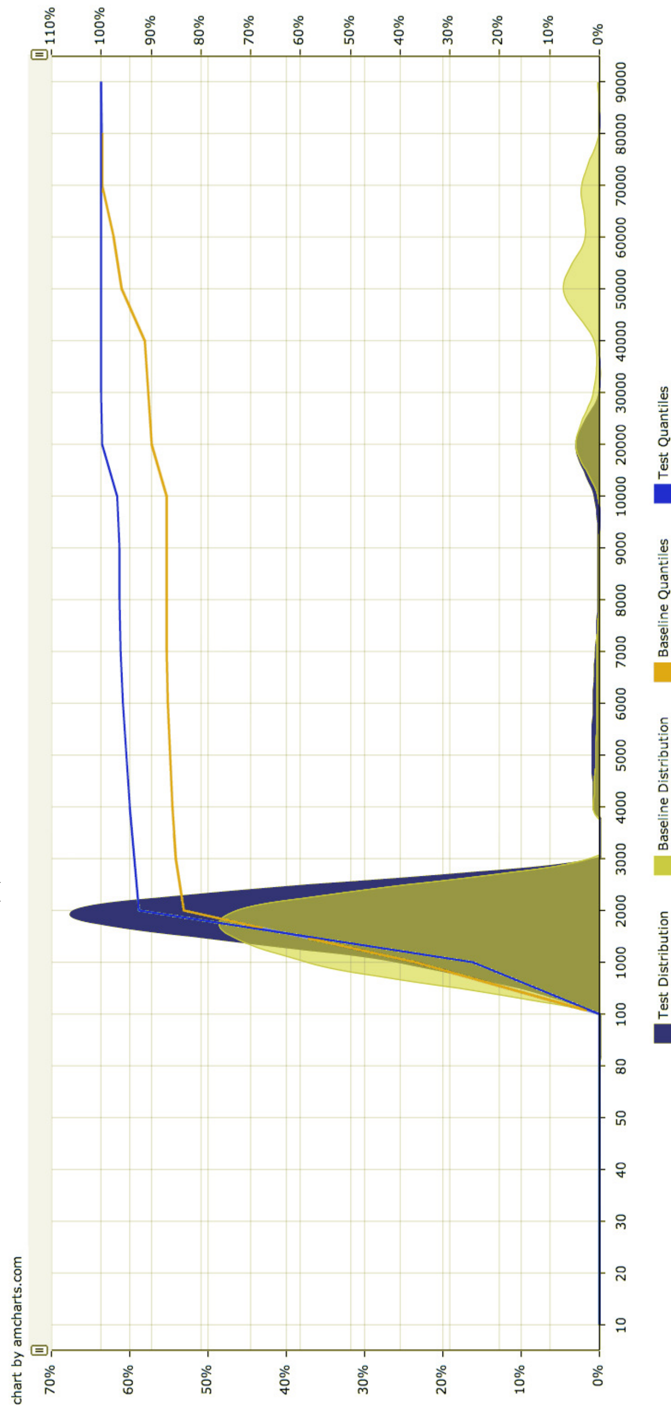
3.3.2 Experimental Results

We conducted two trials, one without any adaptation process and another applying *Zanshin*'s adaptation strategies. The results we extracted from JMeter's output are presented in figures 3.8a and 3.8b, produced by the online service Loadosophia[2]. The *Baseline* trial represents the one without the adaptation process, whereas the one referred as *Test* represents the trial where *Zanshin* controls *ZNN.com*.

Figure 3.8a shows that the response time has been improved by 67.4% after applying the adaptation strategies and the throughput has been improved by 8.7%. While in Figure 3.8b the distribution of the experienced response times is depicted. From the latter we notice that in the case where *Zanshin* is present the distribution of the low response times is higher than in the case where an adaptation mechanism is absent.

To evaluate the efficiency of our approach we measured the failures of the *AwReqs* for every trial. The results have shown that without the use of an adaptation framework AR3 failed 518 times, while with the use of *Zanshin*

[2]https://loadosophia.org



	Baseline Value	Test Value	Difference	Hint
Test Duration:	1 hour, 3 minutes, 59 seconds	1 hour, 4 minutes, 14 seconds	15 seconds	Duration difference 0.4% is small enough, comparison is valid
Average Virtual Users:	387.91	405.552	17.642	VU difference 4.5% small enough, comparison is valid
Average Rate (TPS):	7.10732	7.7234	0.6161	+8.7%, Test served more requests pers second
Average Response Time (ms):	7544	2461	-5083	-67.4%, Test was faster

(a) Summary comparison report

(b) Distribution of response times

Figure 3.8: Experimental results

it failed 408 times but also AR2 failed 214 times. The improvement in the performance is obvious but it came with the cost of not providing high fidelity content for the whole duration of the experiment. The analysts of the system could use these metrics to evaluate their strategies and apply the suitable thresholds.

3.3.3 Discussion

In this subsection we juxtapose the core ideas of the architecture-based approach followed by *Rainbow* and the RE-based approach followed by *Zanshin*.

We start by noticing that both approaches base their adaptation process on a closed loop, where the system monitors its output, detects possible malfunctions and changes its parameters in order to keep fulfilling its mandate. The necessity of the closed loop in software engineering has been pointed out by [BMSG+09] as a tool that will give the opportunity to produce systems based on the principles of *Control Theory*. Another common point of the two frameworks, is that the control is external, which means that the target system does not implement any part of the control loop. In [CGS05] it is mentioned that by delegating the control of the system to an external mechanism, higher generality, cost-effectiveness and composability can be achieved.

The main difference of the two frameworks lies on the different kinds of models they utilise to support their adaptation mechanism. The architectural model of *Rainbow* gives information about the capabilities and the restrictions of the system, which later on will be exploited as operators and adaptation conditions accordingly. Furthermore, basing the adaptation strategies on an architectural model that describes a family of systems makes them reusable to any target system that conforms to the same architecture.

However, having as a starting point the architectural model of the system can result in capturing only low level requirements about it. On the other hand, a requirements model as the one *Zanshin* uses can capture every requirement that comes from the stakeholders. Nevertheless, technical restrictions and properties can be revealed only at a later stage of the requirements analysis process and sometimes important details are overlooked unintentionally. In the next chapter we examine case of requirements that were elicited only when an architectural decision was made first.

We saw earlier that the possible adaptation actions of *Rainbow* are defined by the set of basic operators provided by the target system, e.g., activate server and change fidelity. These operators can be combined in tactics, which are then combined in strategies expressed in Stitch language. These strategies are intended to encapsulate human expertise on specific situations, where external intervention is required to restore a malfunctioning system. On the other hand, *Zanshin* provides two kinds of adaptation: *reconfiguration* and *evolution*. A reconfiguration can either change a parameter of the system (control variable) or switch to an alternative selected for a variant point (OR-refinement on the goal model). The new configuration is informed to the system, which can then take further actions related to this change. It is also important to point out that the ability of self-inspecting in *Zanshin* provides a lot of expression power for its adaptation strategies. However, as it is usually the case in any modelling language, this should be used with care in order not to make models that are very difficult to manage. These modifications are based on the differential relations mined during the system identification process and let the system compose its own adaptation strategies given the holding conditions.

EvoReqs, however, are modelled as Event-Condition-Action (ECA) rules, where the actions are composed of sixteen basic operations [SLAM13].

From these, thirteen are system-specific thus must be implemented in the target system. These operations allow the adaption mechanism, among other things, to retry a given goal, to change the parameters of the system, to delegate the issue to an actor and to relax the awareness requirements (meta-adaptation). In the previous section we managed to express *Rainbow* strategies with *EvoReqs* and Reconfiguration using applicability and resolution conditions. Defining a formal transformation from one approach to the other, though, is not an easy task. The adaptation strategies composed by *EvoReqs* are more close to those of the Rainbow framework written in Stitch as they both capture static administrative operations while the latter gives a more clear representation of the priorities of the objectives using Utility Theory [Fis70].

Regarding the monitoring part (malfunction detection), in *Rainbow* an adaptation is triggered when any invariant in the ACME model fails. An example of invariant is $response\ time < MAX_{RT}$. Thus, if the *response time* gets equal or higher than the maximum allowed, an adaptation is triggered. Instead of invariants, *Zanshin* utilizes *AwReqs* in the requirements model to reason about the status of the target system's operation.

For instance, considering a quality constraint of $responsetime < MAX_{RT}$, an *AwReq* may state that this should be the case in at least 90% of the time. If the percentage goes below that threshold, an adaptation is triggered. We can say that both frameworks are based on the models that they use as a centerpiece for their adaptation, in order to define the variables of the system which should be monitored. However, the variety of *AwReqs* offer a higher level of expressiveness to represent objectives to be satisfied at runtime, than the simple conditions of the architectural model.

Another comparison point is adaptation triggering. In *Rainbow*, it is guided by pre-conditions for the execution of adaptation strategies. If more than one is applicable, the best one is selected according to an aggregate

attribute vector that considers a) the cost-benefit of the tactics in a strategy, b) the weights of predefined criteria, and c) the likelihood of each tactic being applicable. In *Zanshin*, *EvoReqs* have a similar format: there are pre-conditions that define whether a strategy applies or not. If more than one is applicable, the first one is selected (according to the order on which the *EvoReqs* were defined).

However, when *Zanshin* uses reconfiguration, the new values for the system's parameters are defined based on a control-theoretic approach. Differential relations are used to define the impact of parameter changes on the *AwReqs* (benefit). Different adaptation algorithms can be used to select which reconfiguration to perform. In Section 3.2.2, given that we were mirroring the scenario implemented in *Rainbow*, only one parameter (*NoS*) was used in the process and reconfiguration was trivial. We could have, however, included differential relations about *VP1* as well:

$$\Delta\left(I2/VP1\right) > 0 \tag{3.3}$$

$$\Delta\left(I3/VP1\right) < 0 \tag{3.4}$$

These equations represent the fact that an increase in the fidelity level would contribute positively to the success rate of *I2* (3.3) but at the same time decrease the success rate of the *I3* (3.4). Considering these equations together with the ones presented earlier, we can prioritize the relations that involve the same indicator to declare which parameters have greater impact on it. For example, $\Delta\left(I3/NoS\right)[0, maxServers] > \Delta\left(I3/VP1\right)$ would mean that by increasing the number of servers the probability to have high performance (*AR3*) is increasing faster than by decreasing the fidelity. This way, *Zanshin* can provide dynamic adaptation based on control theory principles, while the adaptation process in *Rainbow* is in this sense static.

Finally, we contrast how the two frameworks deal with adaptation fail-

ures. In *Rainbow* a tactic fails when a) its pre-conditions are not satisfied, b) the execution of its operators fail, or c) the result of the tactic is different than expected (which is assessed through post-conditions). These failures are predicted in the strategies, which can request alternative tactics to be executed when a given tactic fails. If all the possible tactics were applied, but the goal of the strategy is not achieved, a termination condition is triggered and the strategy ends. Then *Rainbow* will recalculate which is the most suitable strategy to apply. Similarly, in *Zanshin* when none of the applicability conditions of the *EvoReqs* for a given adaptation is satisfied, the adaptation is aborted. After an adaptation action is performed, but the *AwReq* that triggered the adaptation still fails, the adaptation selection is performed again.

3.4 Chapter Summary

In this chapter we conducted a comparative study between two adaptation approaches, one architecture-based and one RE-based. As a reference point we used the *ZNN.com* exemplar, applying both frameworks to provide adaptation mechanisms according to its described scenarios. Results have shown that both frameworks can provide significant improvement to the system's operation, without any human intervention.

We also performed a side by side comparison of the core elements of both approaches. The outcome of this comparison is that architecture models can capture all the properties and technical restrictions of the target system and by using them as a guide to develop adaptation strategies the reusability of the adaptation mechanism becomes applicable.

More specifically, *Rainbow* captures the human experience and expertise in its strategies and, by applying techniques from decision theory, selects the one that is most suitable. Therefore, the control level of *Rainbow* does

not exceed the one of the human intervention but automates it, offering better reaction time and eliminating human errors.

On the other hand, a requirements model captures more explicitly the objectives of the systems and, with the use of quality constraints, can also express the technical restrictions. However, the exact values of the thresholds of these constraints can be provided either by the instantiation of the architecture model or by the expertise of the system analyst. Moreover, some practitioners may consider that detailed architectural information do not belong in requirements models.

Regarding the adaptation process, the *Zanshin* framework provides higher variability by applying *EvoReqs* or letting the system adjust its parameters through a reconfiguration strategy. In this way, the system relies its adaptation process not only on human expertise but also on well-founded principles of control theory.

Our comparison focuses only on requirements and architectural approaches while behavioural adaptation is ignored. The main reason is the lack of any landmark framework that could be used to express the adaptation strategies the way *Zanshin* and *Rainbow* did. However, in the next chapter we discuss in detail how behaviour can play a significant role in the adaptation process.

In summary, this study revealed the advantages and the vulnerabilities of two well-known approaches in the field of software adaptation. The results suggest that requirement and architectural models should be combined in order to capture every detail of the target system's adaptation needs. The purpose of this combination is to mine all the alternatives that are embedded in the solution and the problem space. The requirements models can provide a broader set of alternatives (e.g., in the case of *ZNN.com*, delegate the video hosting to an external service, such as YouTube), while the architectural models can provide variability in the

deployment of the solution. Moreover, *AwReqs* and parameters can indicate the specific components of the system or variables of the environment that should be monitored, instead of putting probes empirically.

Chapter 4

Designing Adaptation Spaces

> Some problems are better evaded than solved.
>
> C. A. R. Hoare

The results of our comparative study in the previous chapter have shown that the approaches on software adaptation using requirement models can be complementary to those using architecture models, since they focus on different but equally important aspects of the system.

In this chapter we provide an answer to **RQ2** and go beyond this one-dimensional view of adaptation spaces by defining adaptation spaces that accommodate three complementary dimensions. The first dimension captures variability in fulfilling requirements and represents variability in the problem space of the system-to-be, whereas the other two dimensions capture variability with respect to behaviour and architecture. The last two dimensions capture variability in the solution space of the system-to-be, representing how, by whom and in what sequence requirements are to be fulfilled. Together, the three dimensions constitute the adaptation space where an adaptive system searches for alternative reconfigurations.

More specifically, we propose a parametrized model for adaptation spaces that is constituted by a requirements, an architectural and a behavioural dimension. Moreover, we propose a technique for building such models

by adopting a Three-Peaks approach where an adaptation space is defined iteratively by introducing some requirements, deciding on their architectural and behavioural dimensions, and then going back and introducing more requirements, including ones that are determined by architectural and behavioural decisions. This work extends the Twin-Peaks approach [Nus01] which advocates that as the granularity of the requirements grows so does the system's architecture's. In other words, every time a requirement is elicited or further refined, an architectural decision must be taken about what components are going to fulfil it. Taking architectural decisions in parallel with requirements refinement reveals additional constraints and consequently requirements the system-to-be must satisfy. Therefore, an intertwining design and requirement refinement can result in a more complete specification. Finally, to better illustrate and evaluate our approach we use the Meeting-Scheduler exemplar.

4.1 Capturing and exploring variability

In Chapter 1 we motivated the need to introduce variability along all three dimensions – goals, behaviours, architecture – to ensure that the system has a large space of adaptation options in trying to cope with one or more requirements failures. In this section we demonstrate how to elicit and capture behavioural and architectural *control parameters* (CPs) and their impact on system requirements.

4.1.1 Variability in behaviour

The semantics of AND/OR refinements are clear at design time: If goal G is AND/OR refined into sub-goals $G1, ..., Gn$, then the functionality of the system-to-be needs to include functions that fulfil all/at least one of $G1, ..., Gn$.

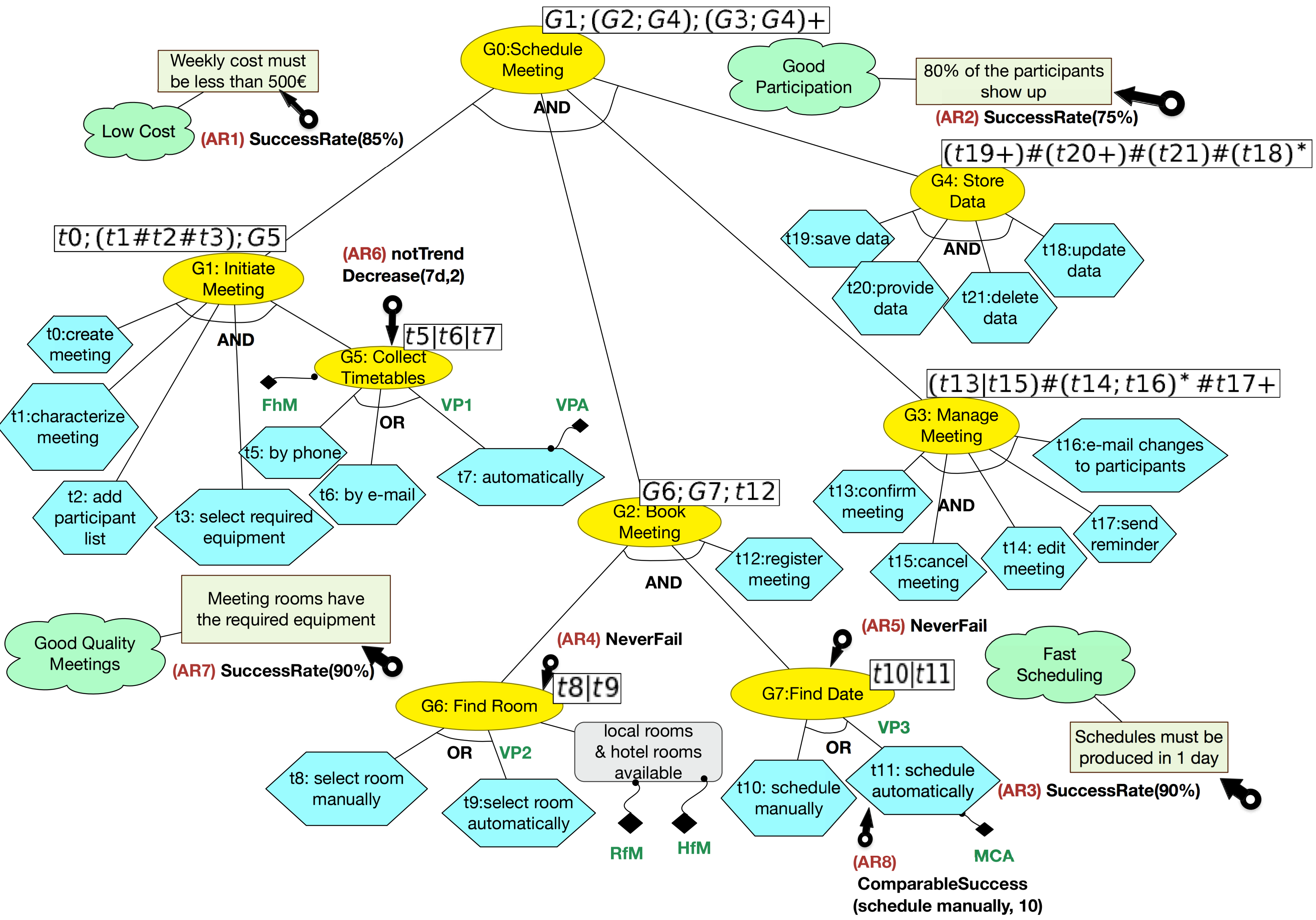


Figure 4.1: Goal model for the Meeting Scheduler case study with flow expressions.

Behaviour talks about the allowable sequences of fulfilment of $G1, ...Gn$ at runtime and we use flow expressions as described in Section 2.1.7. Each sequence needs to include one or more of $G1, ..., Gn$, but not all. So, it can be the case that for an AND-refinement we have sequences that fulfil only some of $G1, ..., Gn$ and for OR refinements we have sequences that fulfil all of $G1, ..., Gn$. We For example, the goal *Manage Meeting*, although all of its tasks must be implemented, *confirm meeting* and *cancel meeting* are actually conflicting and their use cannot coincide in the same execution sequence. Therefore, the | operator indicates that only one of the two is allowed for any one execution of the system as shown in Figure 4.1.

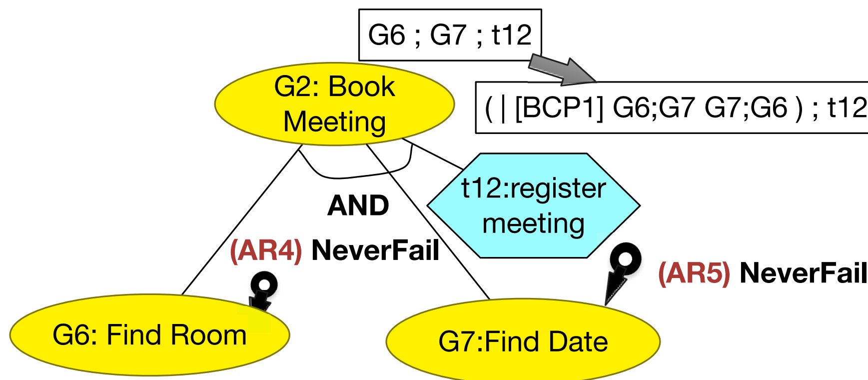


Figure 4.2: BCP from AND-refinement

The ; operator is useful when modelling the behaviour of an AND-refinement and prescribes the order in which sub-goals/tasks must be fulfilled. It is common practice in software design to impose only one possible order, thereby limiting the reconfiguration capabilities of the system-to-be. In our framework, the designer is encouraged to select multiple alternative behaviours for fulfilling a goal. Accordingly, we introduce *behavioural control parameters* (*BCPs*) that are assigned to the goal's behaviour and whose possible values are all the allowed sequences. A *BCP* is defined as $(|[parameter\ name]alt_1\ ...\ alt_n)$, using infix notation for the alternative operator. For example, for the goal *Book Meeting* if the meeting organiz-

ers select a meeting room first and then find a date, participation might be low because of conflicts with participant time tables. If they select the date first and the room afterwards, participation may improve but it is not guaranteed that the selected room will have all required equipment. A *BCP* defined by $(|\ [BCP1]\ G6; G7\ \ G7; G6)$ takes as values the two possible sequences $G6; G7$ and $G7; G6$. Its impact on the requirements is captured by the differential relations $\Delta(I2/BCP1)[G6; G7 \rightarrow G7; G6] > 0$ and $\Delta(I7/BCP1)[G6; G7 \rightarrow G7; G6] < 0$ while the new behaviour for the goal *Book Meeting* is depicted in Figure 4.2.

Another variability factor of system behaviour is related to the multiplicity of the fulfilments of a goal or a task. When there is the option for the system to fulfil multiple times a goal or a task, the designer must consider the impact of this variability on *AwReqs*. For example, the task *t17 send reminder* is performed by the system–to-be and can be executed multiple times if the goal *Good Participation* is failing. To this end, as depicted in Figure 4.3, we substitute when needed the operators `*` and `+` with a *BCP* (in this case named *NoR*) and based on a differential relation such as $\Delta(I2/NoR) > 0$ the adaptation mechanism can adjust its value when *Good Participation* is failing. The range of values of NoR varies from one to five executions of task *t17*.

The repetitive execution of a task or fulfilment of a goal raises the issue of time synchronization. In the previous example, if $NoR = 3$ and all the reminders are sent one after the other within seconds, the outcome is likely to be an unhappy one. Hence, we introduce a *behavioural function wait()* that takes as argument a *BCP* with a range of values related to time units, in this case days. This function is part of the behavioural model as shown in Figure 4.3 and $BCP3$ is defined as $(|\ [BCP3]\ 1day\ 2days\ 3days)$.

Next, we revisit OR-refinements in order to extract additional variability. The traditional perception of these refinements at runtime is that the

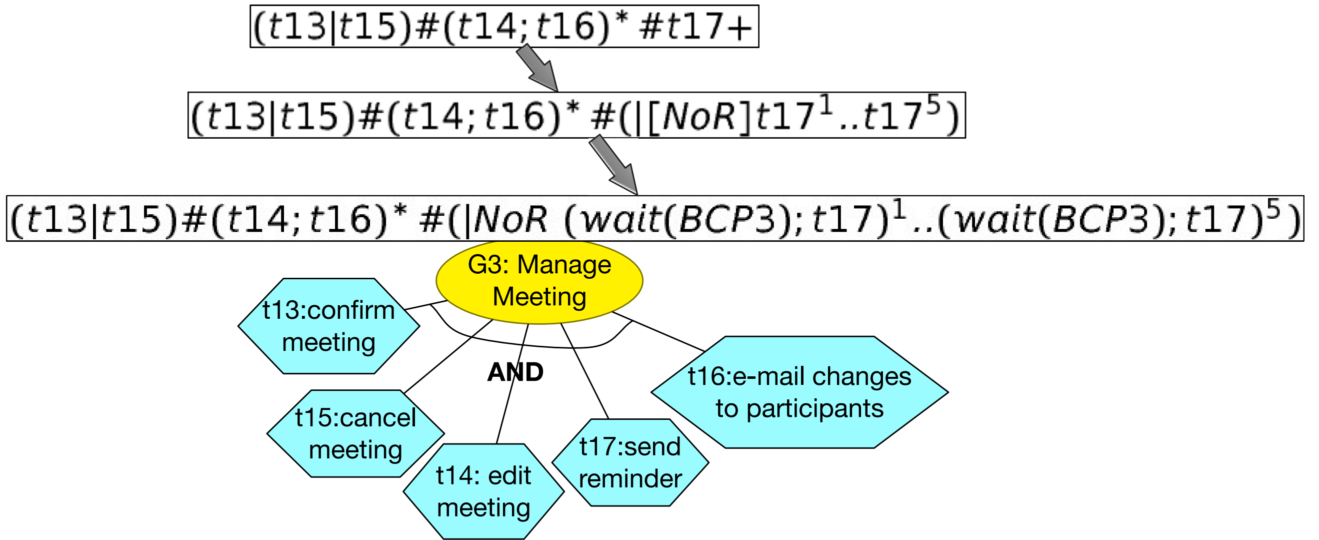


Figure 4.3: BCP from multiplicity operator

satisfaction of any subgoal would lead to the satisfaction of the parent goal. Therefore, the *ReqCPs* associated with an OR-refinement have as candidate value one of the subgoals. The system-to-be though may require in certain occasions the fulfilment of all the subgoals to guarantee the satisfaction of the parent goal. For example, scheduling a meeting requires the fulfilment of the goal *G5: Collect Timetables* that can be achieved by either contacting the participants *by phone*, *by e-mail* or collecting them *automatically* from a common system calendar. However, when one or more of the invited participants do not use the system calendar the third option could harm *AR2*, since these participants will not receive any invitation for the meeting. Dealing with such a situation requires the utilization of all the alternatives under the OR-refinement. This means that participants who do not have an account for using the system's calendar and therefore their timetables must be collected either *by phone* or *by e-mail* while the timetables of the remaining participants can be collected automatically by the system. To capture this additional variability a new *BCP* is introduced defined as (| $[BCP2]$ $VP1$ $t5\#t7$ $t6\#t7$), as depicted in Figure 4.4.

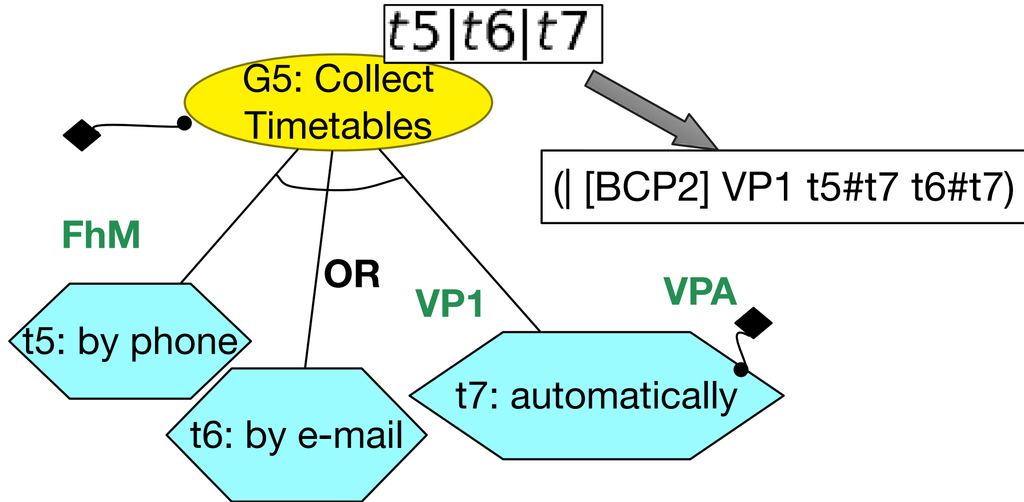


Figure 4.4: BCP from OR-refinement

As a rule of thumb the designers must analyze each AND refinement of the goal model and using domain expertize, whenever the sequence in which the subgoals are fulfilled has impact on one or more indicators, then a *BCP* is identified. Next, the designers must identify how many times each goal or task must be fulfilled in order the parent goal to be fulfilled. If the number is greater than one, it influences one or more indicators and can be put under the control of the adaptation mechanism, the a *BCP* is identified. In the second case, the designers must synchronize the multiple fulfilments of a goal or task with the use of the $wait()$ function.

4.1.2 Variability in architecture

We consider next the third peak, architecture, looking for opportunities to introduce variability. In order to be fulfilled, each goal or task must be assigned to at least one component[1]. For this peak there are two sources of variability. The first is related to each component's multiplicity. Certain components may be instantiated multiple times for requirements to be ful-

[1]Each component must be able to satisfy on its own the assigned goal.

filled. For example, as shown in Figure 4.5, an instance of the component *TimetableCollector* can be associated with multiple instances of the component *Secretary*. The number of instances of the latter, is an adjustable variable that affects the operational cost of the meeting scheduling process ($AR1$), but also how fast the meetings are scheduled ($AR3$). We refer to such variables as *architectural control parameters* ($ACPs$) following the same definition construct as $BCPs$. In this case we introduce the number of secretaries *NoS* parameter defined as (| $[NoS]$ 1..5) that will substitute the abstract multiplicity notation, representing explicitly the presence of a new configuration point. The impact of this ACP on the requirements is captured by the differential relations $\Delta(I1/NoS) < 0$ and $\Delta(I3/NoS) > 0$.

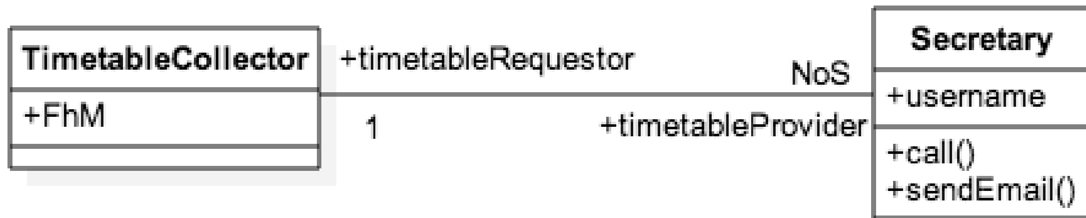


Figure 4.5: ACP for component instance

The second source of architectural variability is related to the selection among multiple candidate components that are assigned with the same goal/task. For the goal *Find Room* we have two candidate software components that are both part of the system and can be used interchangeably. The first component finds the cheapest room reducing the overall cost of the meetings, but does not guarantee that all the required equipment will be present, while the other one finds the best equipped room but might exceed the budget available for scheduling meetings. These two components can be used either interchangeably or concurrently. The concurrent use of both components allows the users select which result is more suitable for them. In specific occasions such as low budget periods, the system may

switch to the exclusive use of the component that provides the best price. Therefore, as shown in Figure 4.6 we add to the architecture model an ACP named $ACP1$ with candidate values all the possible uses of these components, with the following definition (| [$ACP1$] $BestEquipRoomFinder$ $BestPriceRoomFinder$ $BestEquipRoomFinder\#BestPriceRoomFinder$). The shuffle operator indicates concurrent use of the operand components.

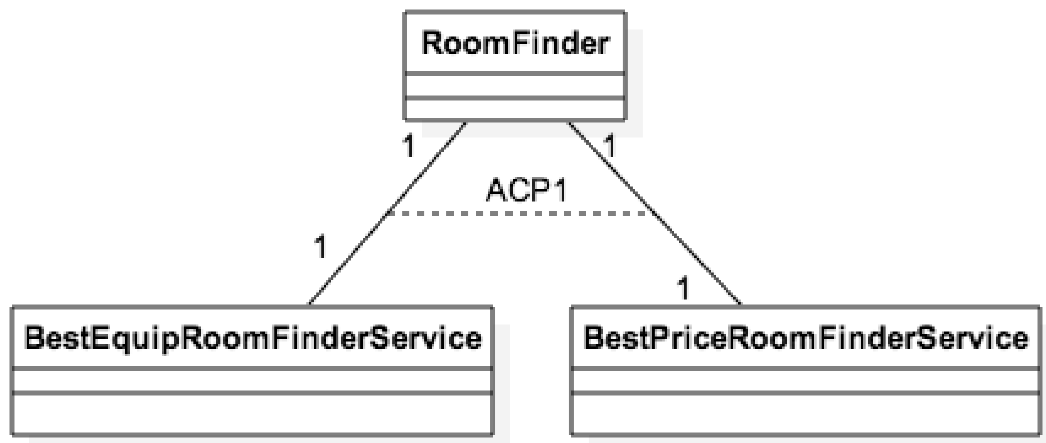


Figure 4.6: ACP for alternative component

As a guideline to the designers, architectural variability lies in relationships of among components, where multiplicity greater than one is present. In such cases, if the number of instances can be controlled by the adaptation mechanism and then an ACP is identified. In practice, components with not a constant number of instances play the role of resources. An active resource is responsible for carrying out tasks (e.g. secretaries) and a passive resource is used as a means of executing a task (e.g. hotel rooms). As it concerns the architectural variability related to alternative components, the designers every time a component is assigned with a goal/task must examine if an alternative component is capable of fulfilling the same goal/task but influence different indicators. However, the financial limitations and the component dependencies must be taken into account.

4.1.3 Variability in the environment

In the previous sections we examined the variability in the three dimensions that constitute the software system that can be controlled by the adaptation mechanism. As we explained in Chapter 1 the environment's uncertainty is one of the main motivations for designing self-adaptive systems. Hence, we need to explore the variables in the environment of the system, that are the driving force for eliciting large adaptation spaces.

Toward this direction, we introduce a domain model, as shown in Figure 4.7. Environmental variability is captured here with a new type of parameter named *environmental parameter* (*EP*).

An *EP* can indicate the number of instances of a domain entity and therefore its multiplicity in the domain model. The difference from architectural multiplicity is that in the case of the environment the adaptation mechanism has no control on the value of the *EPs*. For instance, there is no control on the *number of meeting requests* (*NoMR*) the meeting organizers are sending, neither the *number of participants* (*NoP*) attending a meeting, and therefore these are represented as *EPs*.

The attributes and the operations of domain entities constitute another source for environmental variability. For example, participants may confirm their participation, but in the end not attend a meeting. The *EP percentage of consistency* (*PoC*) captures this, while the *average hotel price* captures the current average cost for reserving a hotel room for meetings.

EPs influence the *AwReqs* in the same manner as *CPs*. However, the adaptation mechanism can only monitor them, identifying undesired situations and change *CPs* to compensate for changes. For example, when the *PoC* is decreased because participants tend to forget meetings they are supposed to attend and the participation is harmed according to the differential relation $\Delta(I2/PoC) < 0$, then the adaptation mechanism can

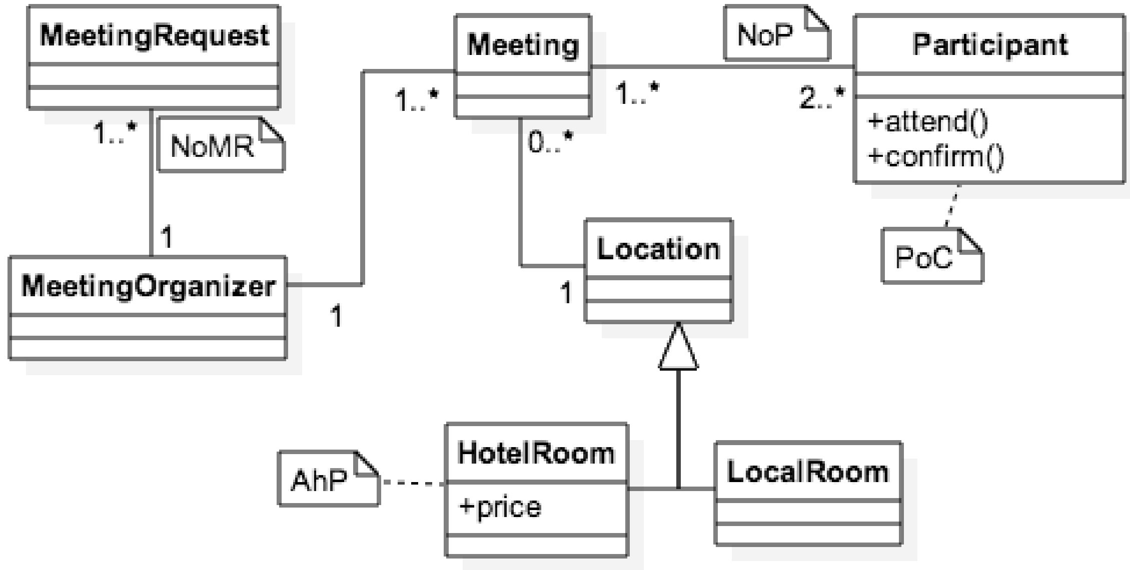


Figure 4.7: Domain model for the Meeting Scheduler environment

increase *NoR* to compensate.

4.2 A Three-Peaks modelling process

The modelling process for Three-Peaks models is depicted in Figure 4.8. It guides the elicitation of all elements of a Three-Peaks model, including control parameters. Our process is iterative and intertwined, analyzing and expanding problem and solution spaces simultaneously.

The process starts by getting as input a goal or a task, which initially will be the root goal such as *G0: Schedule Meeting*. The next step is to identify if there are any *AwReqs*, softgoals or domain assumptions related to the input. Then, if the input is a goal, it is refined into subgoals, otherwise the requirement and behaviour analysis are skipped. The designers, along with domain experts, examine what needs to be fulfilled and how, starting from eliciting parameters required for that inserted goal to be satisfied, such as how many conflicts are allowed before finding a date or if the system can view private appointments. These parameters are *ReqCPs* and their values

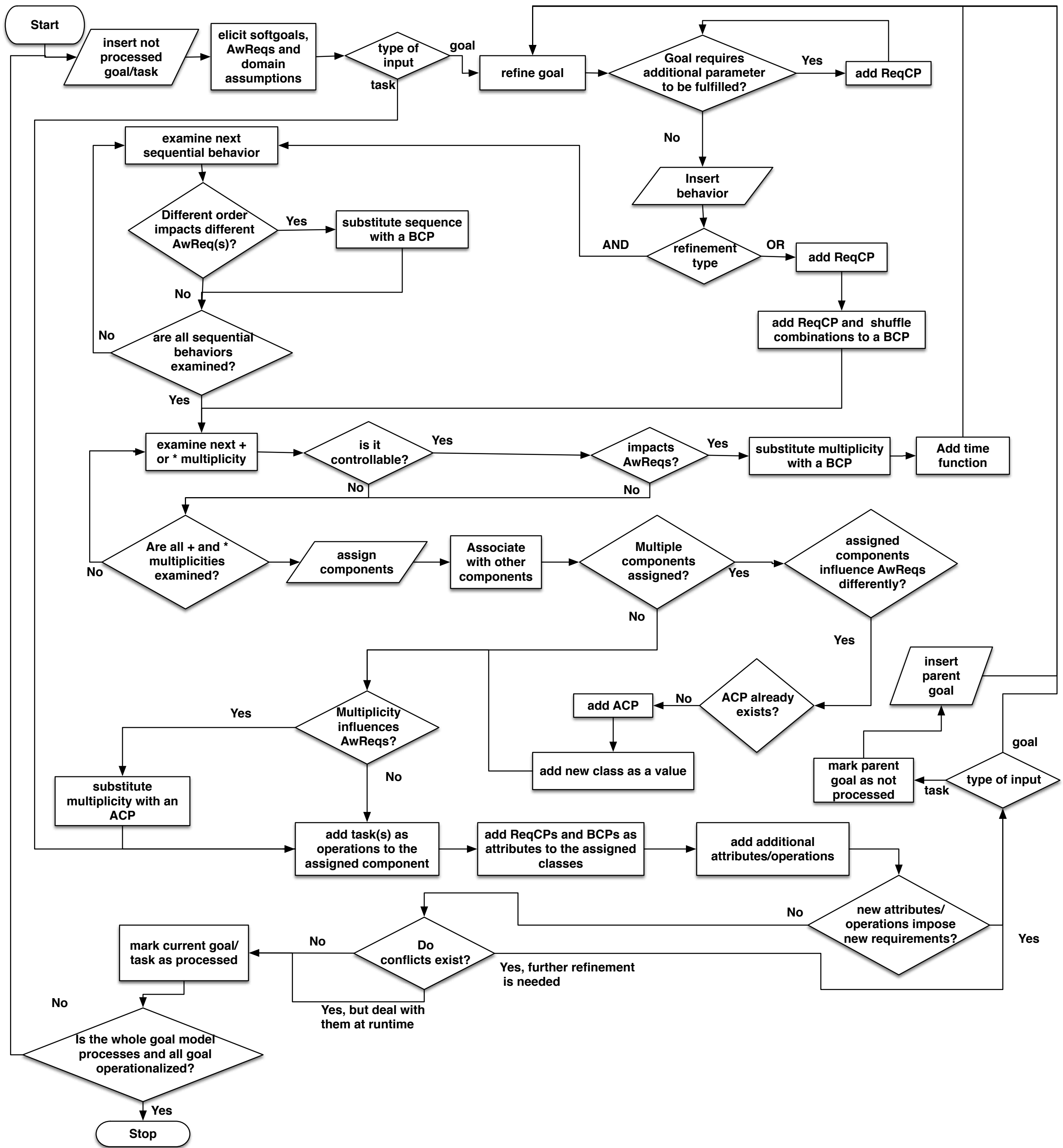


Figure 4.8: The Three-Peaks process as a flowchart

may vary during alternative executions of the system.

Continuing the analysis of how a goal can be fulfilled, designers provide an initial behavioural model using the notation introduced in Section 2.1.4. If the goal is OR-refined then each subgoal becomes a candidate value for a *ReqCP* such as $V1 - V3$ in Figure 4.1. Then, the behavioural model is refined by adding a BCP with range of values according to existing *ReqCP* and shuffle combinations of the refinements as in Figure 4.4. In the case of AND-refinement, the order in which the operands of sequential behaviours (the parts of the model that include only the ; operator) is examined. If a different order of the operands implies influence to different *AwReqs* a new *BCP* is introduced with range of values, all the potential orders. Concluding this iteration of behaviour analysis, the process examines every $*$ and $+$ operators in order to substitute them with a *BCP*, if needed, as described in Section 4.1. In that case also the *wait(BCP)* function with its own *BCP* is added. The last step leads to a new refinement of the goal since a *wait* task is added as a refinement of the examined goal.

Moving to the architecture peak, designers associate the input goal or task to one or more components of the architecture. This determines who is responsible for the satisfaction of the goal or task. When more than one component is assigned, an *ACP* is added and can be tuned by the adaptation mechanism at runtime in order to activate the most suitable component or a combination of them for fixing failing requirements. Next, if the new component can be instantiated multiple times at runtime and this number has impact on the *AwReqs* while under the control of the adaptation mechanism, the associated multiplicity is substituted with an *ACP*. Then, the assigned components get as attributes the *ReqCPs* and *BCPs* of the goal, as they must be aware of what behaviour must follow and what are the values of these parameters. Once the previously elicited variability has been embedded in the assigned component, the designers

of the architecture, provided that the goal is fully or partially operationalized, add the tasks produced by the refinement as operations. Finally, the designers may provide additional attributes and operations to the component of more technical nature that are not related neither to requirements nor to behaviour. For every new attribute or operation, the process must investigate whether there is need of adding new requirements. If the initial input was a goal then it is refined again, but in case of a task then it is the parent goal that must be processed again.

The last step of the process inspects if the current set of configurations is able to guarantee the the satisfaction of all the *AwReqs* related to the investigated goal under any possible environmental condition. In case there are situations where the system is not able to guarantee success of all the related *AwReqs*, then two actions can be taken: a) perform further refinements, finding new *CPs*, goals or tasks; or b) deal with conflicting requirements, using the conflict resolution mechanism that we discuss in detail Chapter 5. When all goals and tasks are processed and every goal is operationalized, the process terminates.

As the designers perform the Three-Peaks process it is very important to keep track of all the differential relation between the elicited *AwReqs* and *CPs*. An ideal result of the process would be a system where every *AwReqs* has a *CP* that if its value changes to improve one indicator it will not decrease another. In other words, every indicator can be controlled independently, by at least one *CP*. Of course, even such an ideal design cannot eliminate the possibility of conflicts. The reason is that every *CP* is bounded and therefore, when the upper or lowest value is reached, the adaptation mechanism might not be able to tune it in order to fix a failing indicator. This is the reason we require large adaptation spaces and design adaptation mechanism to resolve conflicts among requirements that we present in detail in the next chapters.

4.3 Evaluation

Following the Three-Peaks process presented in the previous section we produce a goal model with annotated behaviour (Figure 4.9) and an architectural model (Figure 4.10) which includes several additional parameters over what was presented in Section 2.1.

More specifically, six new *CPs* are derived from the behavioural model ($BCP1 - BCP5$ and *NoR*) and two from the architectural model ($ACP1$ and *NoS*). Moreover, the Three-Peaks process resulted in eliciting three additional tasks ($t22$, $t23$ and $t24$). The task *t23: be online* along with $AR9$ are result of the attribute assigned to the component *Database* that is responsible for the goal *G4: Store Data*. This prescribes that the status of the *Database* must be monitored to ensure that it is constantly online. The task *t24:wait* is introduced by the assigned behaviour to goal *G3: Manage Meeting*. Finally, the task *t22: do meeting online*, has been introduced to resolve situations where there are few suitable dates due to many conflicts among participants, and there are not enough available rooms.

Not taking into account the holding conditions of the environment carries the risk of choosing the wrong adaptation for fixing failing requirements. For instance, consider the case where participants forget to attend their meetings, resulting in the failure of $AR2$. The adaptations offered by the original goal model (not generated by the Three-Peaks process) for fixing $AR2$ are either to start viewing private appointments of participants by setting *VPA* to true or decreasing *MCA* allowing fewer conflicts. Neither one anticipates the real cause of the failure. The Three-Peaks model though offers the parameter *NoR* that increases the number of reminders thereby tackling the source of the problem, and capable of increasing the success rate of $AR4$.

Another case where requirements-only variability proves to be insuffi-

cient concerns the room selection by the meeting organizers before or after finding a date for their meeting. Each of the alternative orders works well in different contexts. Selecting room first guarantees good quality meetings, since the meeting organizers select a room that can provide all the required equipment, assuming that the invited participants are available the same date the room is available, otherwise the success rate of $AR5$ is at risk. On the other hand, when meeting organizers select date first, it is more likely that they will find a date convenient to most invited participants, but a sufficiently equipped room might not be found in periods with high workload for the meeting scheduler, decreasing the success rate of $AR7$. Using behavioural variability, an adaptive meeting scheduler executes the order that complies with the existing context by tuning $BCP1$. Moreover, to maintain the equilibrium between the success rate of $AR1$ and $AR7$ when the system selects rooms automatically the system can use either a component that finds the cheapest room available or the best equipped respectively, exploiting architectural variability and more particularly $ACP1$.

The previous failure scenarios show that the high variability models of the Three-Peaks process can handle better changes in the system's environment where the requirements-only model would provide ineffective adaptations. A limitation of our approach is that dependencies among *CPs* are not captured. For instance, it makes sense for *MCA* to be changed only if the value of $VP3$ is set first to "schedule automatically". In order to alleviate this obstacle, we are planning to extend our notation in order to capture this kind of constraint. Another limitation on the scalability of our proposal is that for every variable introduced into the model, its impact on all *AwReqs* must be examined.

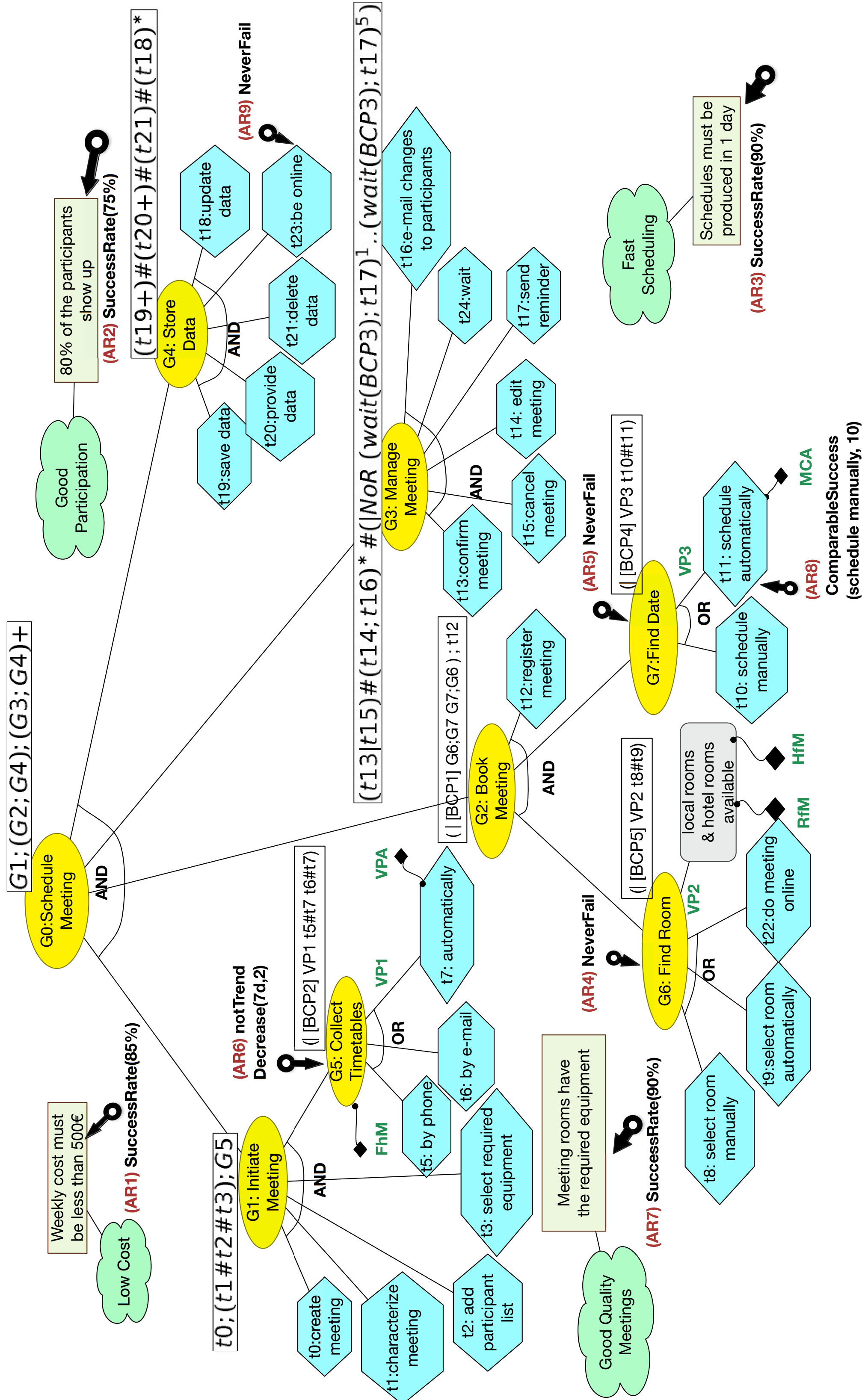


Figure 4.9: The goal model after the Three-Peaks process

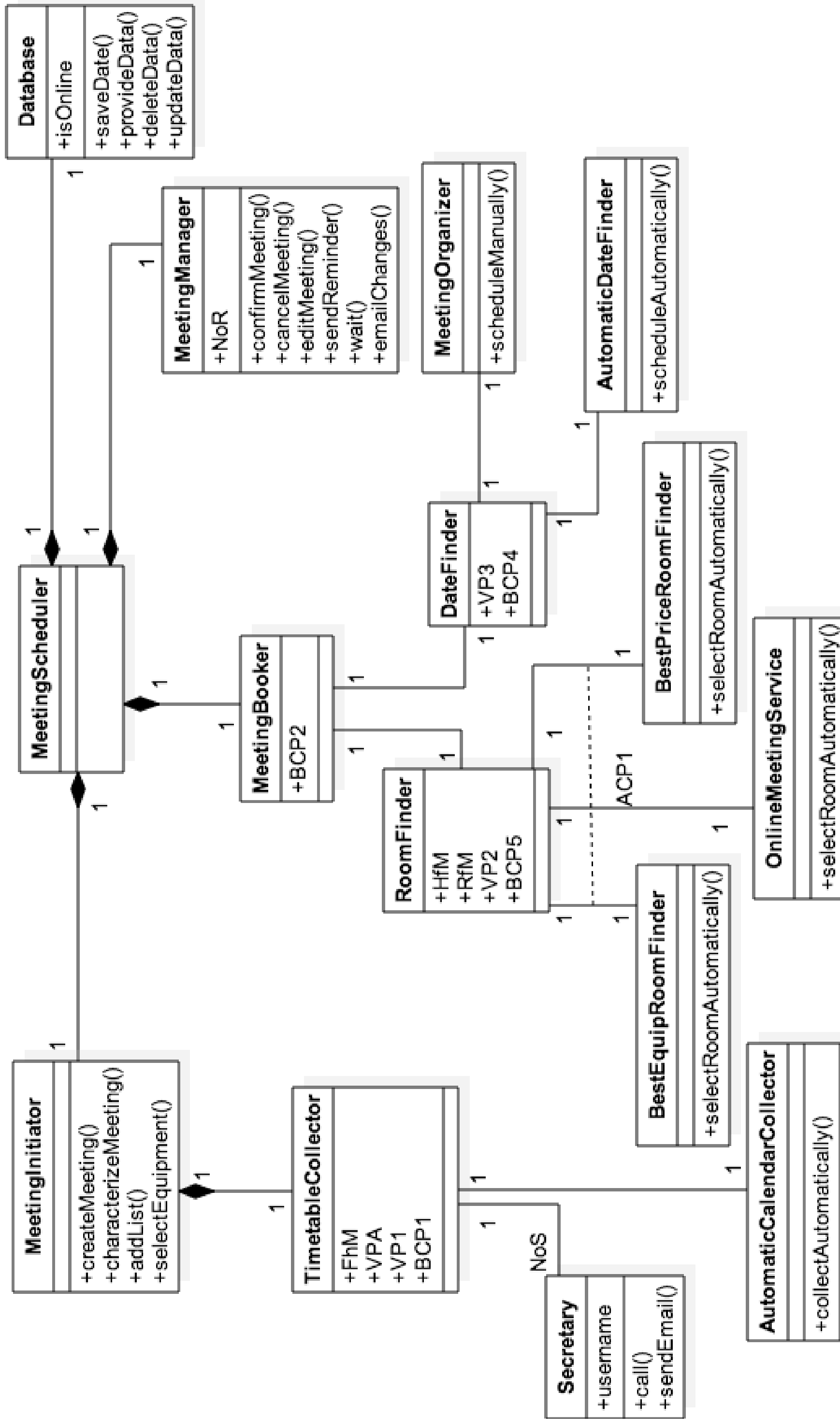


Figure 4.10: The architecture model after the Three-Peaks process

4.4 Chapter Summary

In this chapter we propose a systematic process for extracting incrementally variability from goal models. The source of variability lies in the three peaks of a software system: requirements, behaviour and architecture. We investigate how variability can be elicited along each peak, introducing behavioural and architecture control parameters and how to model environmental variability. We also present a Three-Peaks process to derive incrementally high variability requirements, behavioural and architecture models. Finally, we have evaluated our models through execution scenarios of the meeting scheduler exemplar, showing that offering adaptations along three peaks enables the system to handle more failures.

Chapter 5

Qualitative Adaptation for Multiple Failures

> Things which matter most must never be at the mercy of things which matter least.
>
> Johann Wolfgang von Goethe

In this chapter we propose an adaptation mechanism that can handle multiple failures (i.e., multiple failing requirements). This mechanism is constructed under the assumption that only qualitative information between control parameters and indicators is available, providing an answer to **RQ3**. As opposed to *Zanshin*, which treats failures sequentially, our proposal considers at the same time all failing requirements and attempts to select an adaptation that is coherent in the sense that it reduces the overall degree of failure, taking into account priorities among requirements. Our proposal supports the definition of Adaptation Requirements provided by stakeholders. For example, an adaptation requirement may state that the adaptation should be conservative in that it does not change parameters in a way that could harm non-failing requirements. Such adaptation requirements are taken into account as the adaptation mechanism considers iteratively current failures, selects an adaptation, applies it, and observes results. The ultimate goal of this approach is to handle depen-

dencies among requirements in a way that is consistent with stakeholder expectations about the adaptation process itself.

5.1 Requirements for Adaptation

Handling multiple requirements failures requires trade-offs. To achieve this, we extend *Zanshin* in a way that it can dynamically put together adaptation strategies using priorities over the requirements as criteria for resolving runtime conflicts among requirements that could not be eliminated at design time. We also propose the specification of *Adaptation Requirements* (*AdReqs*) . These requirements are defined by reusing concepts of the *Zanshin* framework, namely *AwReqs* and *EvoReqs*.

5.1.1 Prioritizing Requirements

Requirements are prioritized in order to support the selection among alternative adaptations during the adaptation process. If R, R' are both failing, it is important to know which of the two has higher priority. To make our adaptation mechanism more precise in dealing with failures, we actually require information on how much higher priority does R have over R', a little or a lot. Moreover, we need to define policies about dealing with conflicts even in cases where not every involved requirement is failing. For example and adaptation plan to fix a failing requirement R might threaten a non-failing requirement R' of much higher priority. Hence, the adaptation mechanism should be instructed if taking the risk of fixing a failing requirement but harming a non-failing one is acceptable and if so, under which circumstances.

Given such information, we can adopt the Analytic Hierarchy Process (AHP) [Saa80] which has been proven to be one of the most effective methods to accurately prioritize objectives [KWR98]. Other applications of the

AHP prioritize requirements as a means to select those that are more important to be implemented [KR97]. In our case the prioritization is useful for selecting which requirement failure should be fixed and which one should not because it would create further failures that should lead to changes to other requirements.

The process for prioritizing the indicators of the system's *AwReqs* includes the following steps. First, after the system identification process is carried out using the Three-Peaks process where the qualitative relations among indicators and the parameters are elicited. Of course, as we mentioned in the previous chapter, it is not always possible to resolve every conflict with the use of the Three-Peaks process. We remind that by the term 'conflict' we mean that two indicators are influenced by the same parameter in opposite directions. For example, if $I1$ and $I2$ are both influenced by parameter P_4 and $\Delta\,(I1/P4) > 0$ and $\Delta\,(I2/P4) < 0$ their differential relations mean that if $AR1$ and $AR2$ are failing and we can treat them only by tuning $P4$ then we cannot fix both of them. Then, by using the scale presented in Table 5.2 we compare all the pairs of indicators and assign a value to each pair. For the purpose of illustration consider that we have four indicators $I1$, $I2$, $I3$ and $I4$ and the result of the pairwise comparisons is shown in Table 5.1.

-	I_1	I_2	I_3	I_4
I_1	1	3	1/5	1
I_2	1/3	1	7	1/5
I_3	5	1/7	1	1/5
I_4	1	5	5	1

Table 5.1: Pairwise Comparison Values

For a most effective use of the given scale we apply a set of heuristics that works as a guideline for the stakeholders in order to assign accurate

values. These heuristics are the following:

Heuristic 1: Indicators associated with hard-goals are preferred over those of soft-goals.

Heuristic 2: Indicators of hard-goals that are closer to the top-goal are preferred over lower level ones.

The purpose of these heuristics is to give higher priority to to the functional integrity of the system over satisfaction of the non-functional requirements. The process continues by calculating the eigenvalues and then normalizing sums of rows. The final result is shown in equation 1.

$$\frac{1}{4} \cdot \begin{pmatrix} 0.87 \\ 0.75 \\ 0.84 \\ 1.45 \end{pmatrix} = \begin{pmatrix} 0.22 \\ 0.18 \\ 0.21 \\ 0.36 \end{pmatrix} \begin{pmatrix} I1 \\ I2 \\ I3 \\ I4 \end{pmatrix} \tag{5.1}$$

We have now assigned weights over each indicator that represent their value and we have a numerical guide to perform comparisons when needed. For instance, in the case where the system has to choose between fixing either $I1$ and $I3$ or $I4$, even if $I4$ is ranked higher than the other two their aggregated weight is higher and therefore should be preferred.

5.1.2 Adaptation Requirements

Our proposal includes a component that, given a requirements model and differential relations among indicators and *CPs*, is able to dynamically compose adaptation strategies that can handle multiple failures. As a first step we prioritized the indicators of the target system with weights that also measure their overall contribution to the correct operation of the system. This, combined with the qualitative relations from the system identification process allows the extended *Zanshin* to automatically compose

Relative Intensity	**Definition**	**Explanation**
1	Of equal importance	The two indicators are not conflicting
3	Slightly more important	Experience and judgement slightly favours one indicator over the other
5	Essentially more important	Experience strongly favours one indicator over another
7	Very much more important	An indicator is strongly favoured and its dominance is demonstrated in practice
9	Extremely more important	The evidence favouring one over the other is of the highest possible validity
2,4,6,8	Intermediate values	When compromise needed
Reciprocals: If indicator I has one of the above numbers assigned to it when compared with requirement I', then I' has the reciprocal value when compared with I.		

Table 5.2: Scale For Pairwise Comparisons

adaptation strategies, reconfiguring the system and maximizing the value of the satisfied indicators.

Our framework deals with the following kinds of failure:

- *Single Failure (SF):* Only one indicator is failing and needs to be fixed.
- *Multiple Independent Failures (MIF):* Many indicators are failing and either they do not have any common parameters or the common parameters have the same monotonicity with every failing indicator.
- *Multiple Dependent Failures (MDF):* Many indicators are failing and all or groups of them share parameters with opposite monotonicity.
- *Priority Conflict (PC):* A failing indicator can be fixed only by tuning a parameter that harms a non-failing indicator of higher value.

- *Synchronization Conflict (SC):* An indicator is currently being treated and in the meantime a new failing indicator requires to tune a parameter that has a negative impact on the first one.

In Control Theory systems with multiple goals that present such conflicts are referred as coupled MIMO systems [ADH+08]. A common technique to tackle the problem of coupled goals is Model Predictive Control (MPC) which is described in detain in Chapter 7. However, MPC requires the existence of an analytical model that describes the relationship between control parameters and indicators and is not always available in advance. Therefore, a trade-off mechanism that can perform with qualitative information is required. In this work we exploit the capability of *Zanshin* to achieve conflict resolution and minimize the error caused by failing *AwReqs* with the use of *EvoReqs*. More specifically, *EvoReqs* are used as a compensation mechanism during the the trade-offs. When a conflict arises between two goals, the most important one is fixed whereas the other is either relaxed or suspended, until the system recovers.

Toward this direction, we model in the extended *Zanshin* additional requirements that refer explicitly to the adaptation process, rather than the base-system, named Adaptation Requirements (*AdReqs*). Figure 5.1 presents a simple goal model that prescribes how conflicts should be resolved during the adaptation process. The task *Adapt Conservatively* instructs the framework not to harm non-failing *AwReqs* while fixing the failing ones. The alternative task *Adapt with Compensation* represents the compensation mechanism we mentioned earlier. More specifically, the adaptation framework is allowed to harm a non-failing *AwReqs* of higher value for fixing another one, only if there is a possible action that would increase the indicator of the *AwReq* being harmed. Along the same lines, *Adapt Optimistically* does not consider priority conflicts as hazards for the base system because there is the assumption that the non-failing *AwReqs*

are tolerant enough to a potential negative impact.

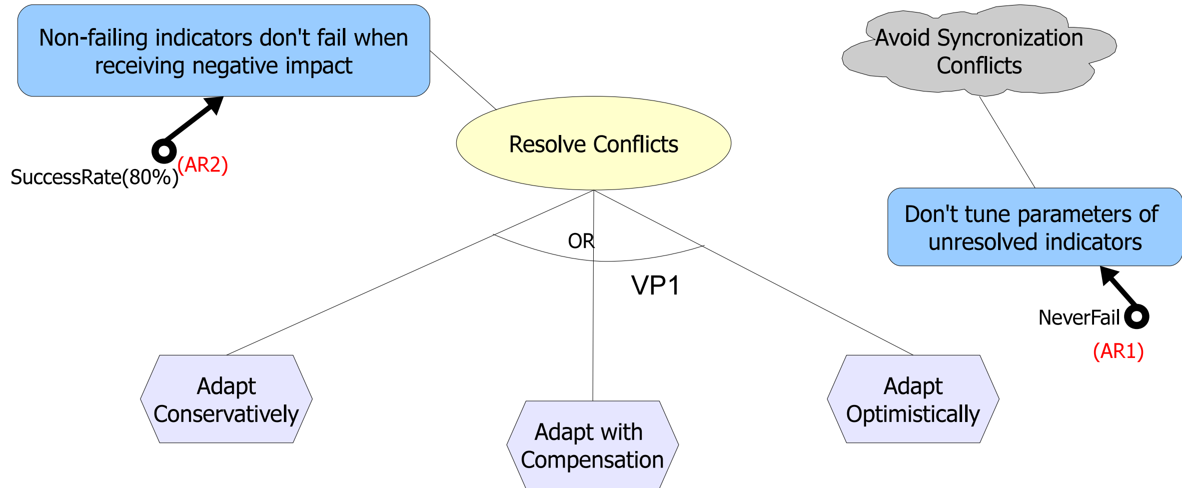


Figure 5.1: Adaptation Requirements Goal Model

To monitor the success/failure of requirements for the adaptation process depicted in Figure 5.1 we use *AwReqs*. In this case $AR2$ imposes the constraint that non-failing *AwReqs* should not fail when receiving negative impact. As we would do for any goal model of a target system we define an adaptation strategy to overcome failures of this Awareness Requirement. Given the differential relation $\Delta\,(I2/VP1)\;[AdaptConservatively \rightarrow Adapt\; withCompensation \rightarrow AdaptOptimistically] < 0$ (the arrow identifies toward which direction the enumeration value grows [SLM11]) the strategy switches among the possible values of the parameter $VP1$.

```
AwReq AR2: Non-failing indicators do not fail when receiving
   negative impact
-Checked at: every 5 minutes
-Adaptation Strategy 3.1: ChangeParam(VP1, Adapt with
   Compensation)
-Applicability Condition: this is the first failure
-Adaptation Strategy 3.2: ChangeParam(VP1, Adapt with
   Compensation)
-Applicability Condition: AS3.1 applied last, more than 5 minutes
    ago
-Adaptation Strategy 3.3: ChangeParam(VP1, Adapt Optimistically)
```

```
-Applicability Condition: no failure for more than 1 hour
```

Listing 5.1: Adaptation Strategy for Adaptation Requirements

Independently of the value of parameter $VP1$, the adaptation framework is required to perform trade-offs among indicators of failing and non failing *AwReqs*. For some indicators there is a parameter to change in order to be brought it closer to fulfilment. For others there will not be any, at least ones that do not harm higher priority indicators. Hence, when potential conflicts are detected during system identification process *EvoReqs* operations (e.g. abort, retry, replace etc) should be assigned to every indicator that may conflict with others. For instance, for the Meeting Scheduler, the goal model of which is depicted in Figure 5.4, $AR6$ and $AR10$ are both dependent on the parameter FhM through the differential relations $\Delta\,(I6/FhM) > 0$ and $\Delta\,(I10/FhM) < 0$. Consequently, if $AR6$ has higher priority than $AR10$ and $AR10$ fails, but the framework applies conservative adaptation, a predefined *EvoReqs* operation for $AR10$ could be $Replace(AR10_1day, AR10_2days)$. This way we acquire compensation in accordance with requirements set by the stakeholders.

Given the fact that the changes to parameter values do not take effect immediately and in the meantime more failures may take place, it is important to apply a form of synchronization to the adaptation process. In Figure 5.1 the $AvoidSynchronizationConflicts$ and the $AR1$ satisfy this need. This goal states that when a failing indicator is being fixed no parameter can be tuned in a way that will affect the failing indicator negatively. For example, if AR8 is failing and the adaptation framework increases RfM to fix it. However, this change may require significant time to take effect and before that happens, $AR2$ fails. In order to fix $AR2$ the parameter $VP2$ must be decreased, but that would affect negatively $AR8$ before the latter has been fixed. To avoid such situations that could lead the adaptation process into non-converging adaptations, we set as a

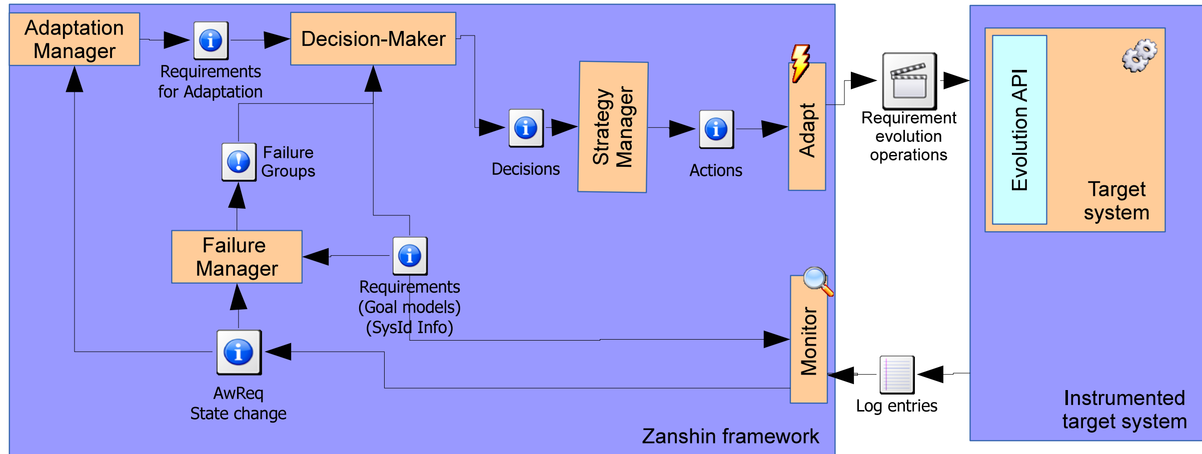


Figure 5.2: *Zanshin* Architecture

requirement not to apply changes that affect unresolved indicators for the sake of fixing other failures.

5.2 Adaptation Process for Multiple Failures

In the previous section we presented a set of features such as prioritization and compensation mechanisms that can help us handle simultaneously multiple failing indicators and compose dynamically adaptation strategies. This section describes the additions to the *Zanshin* framework that implement these features and explains the steps that the adaptation process carries out.

Figure 5.2 depicts the conceptual architecture of the extended *Zanshin* framework. A *Monitor Component* examines the log files that are produced at runtime from the base system; if any failing *AwReqs* are detected the *Failure Manager* and the *Adaptation Manager* are informed. The *Failure Manager* groups failing indicators according to the presence of conflicts with other indicators and informs the *Decision-Maker* component. The *Adaptation Manager* is responsible for configuring the adaptation process by monitoring the requirements model such as the one in Figure 5.1. When

an *AwReq* fails, it selects an adaptation strategy for this failure. Then, it informs the *Decision-Maker* about the selected parameter values of the adaptation process goal model in order to perform the trade-offs accordingly. The *Decision-Maker* exploits input from other components and the indicators' value derived from the AHP to select which indicators should be tuned and which should be compensated. The *Strategy Manager* converts decisions to adaptation strategies by putting together all the required operations. Finally, the *Adapt* component executes the operations that are prescribed in the adaptation strategy.

To give a better understanding of how the framework operates, the diagram of the Figure 5.3 presents the steps of *Qualia+* the adaptation process which is followed by the extended *Zanshin*. The steps are the following:

1. All the *AwReq* failures are collected by the failure manager;
2. The indicators of the failing *AwReqs* are separated with criteria related to the conflicts they may be part of;
3. The decision-maker exploiting the differential relations and the values assigned to each indicator resolves any conflicts that may exist by deciding what action should be performed;
4. The values for the selected parameters are calculated;
5. The values of the selected parameters are changed and the *EvoReqs* operations for the indicators that cannot be treated are executed;
6. The framework waits for the changes to take effect;
7. After the wait time the indicators are evaluated again;

8. In each cycle, the process learns from the outcome of this change and the recorded data could be used to derive quantitative relations among indicators and parameters;

9. Finally, if there are no more failing *AwReqs* after the evaluation the process terminates successfully.

10. Otherwise, the framework will look for violations on the requirements for the adaptation process and if any are found the defined adaptation strategy will be executed.

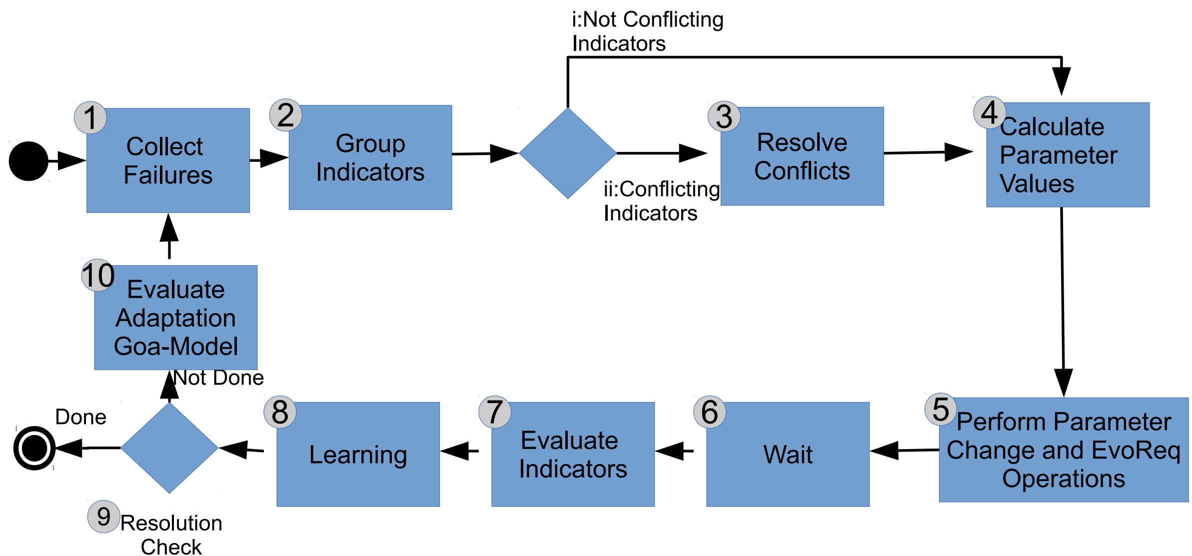


Figure 5.3: *Zanshin*'s Adaptation Process

The strategy manager composes a new strategy with all the actions that will apply the new values to the parameters. These actions are executed by Requirement evolution operations on the target system. The framework waits for an amount of time for the changes to take place and then evaluates the indicators that were treated. There is a step that the framework is

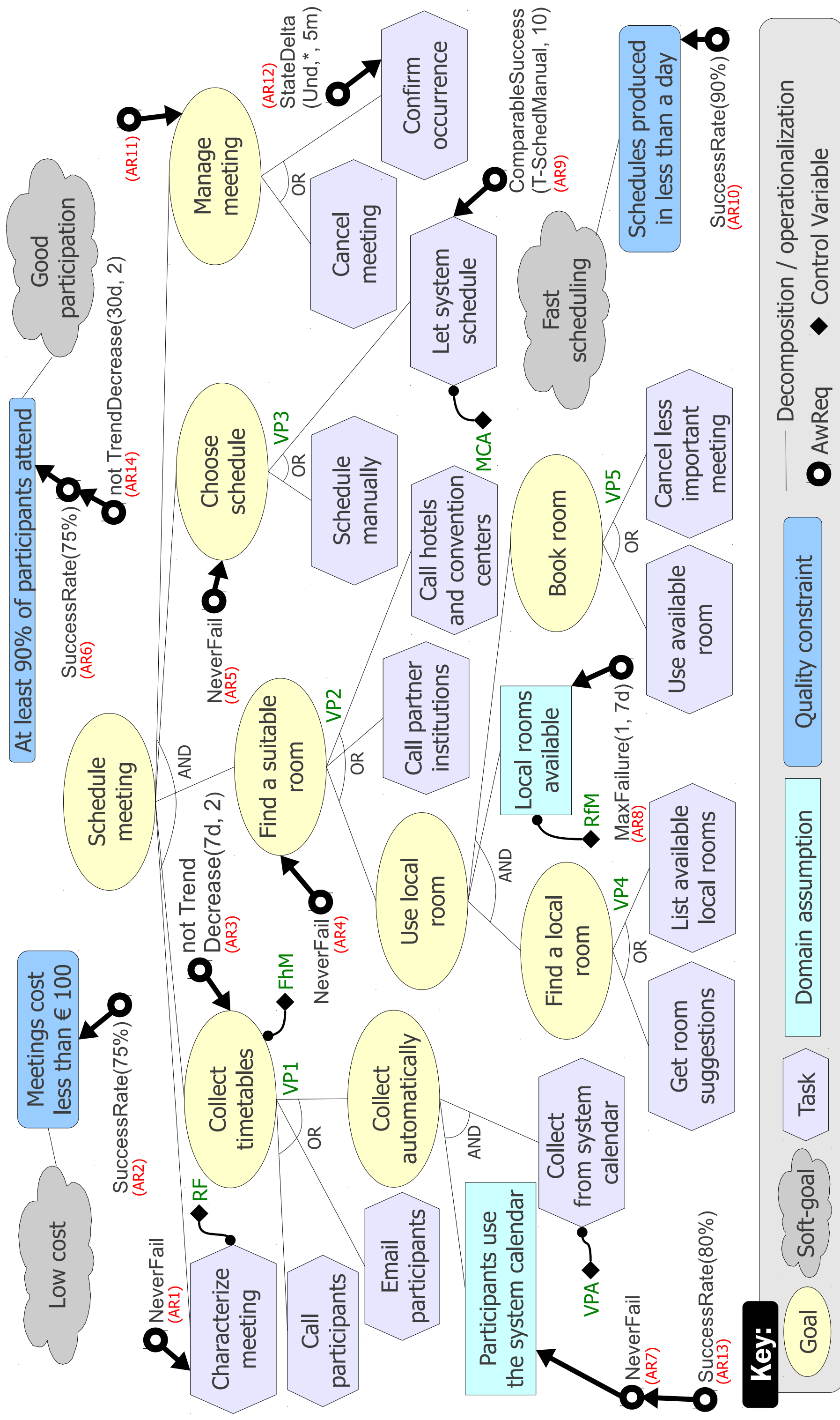


Figure 5.4: Adaptation Requirements Goal Model [SS12]

performing learning in order to derive quantitative relations among parameters and indicators, but it is part of our future research agenda and is not examined in this thesis. Finally, the algorithm terminates if all the failing *AwReqs* are fixed and, if not, the policy manager controls the status of of the adaptation requirements and switches policy if there are any failures.

5.3 Evaluation

This section explains how the *Qualia+* mechanism works through the Meeting Scheduler case study we introduced in Chapter 2. Then we demonstrate several cases that the older version of *Zanshin* would not be able to handle as effectively as the extended one does.

5.3.1 Meeting Scheduler Exemplar

The first step for building an adaptive system that will be managed by our proposed framework is to perform system identification and elicit the differential relations among *AwReqs* and the parameters of the system's goal model. From the goal model of the Meeting Scheduler depicted in Figure 5.4 we elicit a set of differential relations presented in Table 5.3. We then apply AHP as discussed earlier and show the result in Table 5.4. The final step for having all the prerequisites for our frameworks input is to assign to each *AwReq* an *EvoReq* operation to be executed in case it cannot be fixed by changing a parameter value due to the presence of conflict(s). For our case study the assigned *EvoReq* operations are listed in Table 5.5.

According to [SLAM13] we state that some *EvoReq* operations can act either at instance level or class level. For example, when a requirement R is replaced by a requirement R' at an instance level, it means that future runs of the base system will use R, not R'. On the other hand, a class-level

$$order\,(RF) : listonly \prec short \prec full \tag{5.2}$$

$$order\,(VP2, AR10) : partner \prec hotel \prec local \tag{5.3}$$

$$\Delta\,(I1/RF) < 0 \tag{5.4}$$

$$\Delta\,(I2/RfM) < 0 \tag{5.5}$$

$$\Delta\,(I2/VP2) < 0 \tag{5.6}$$

$$\Delta\,(I3/FhM) < 0 \tag{5.7}$$

$$\Delta\,(I4/RfM) > 0 \tag{5.8}$$

$$\Delta\,(I4/VP2) > 0 \tag{5.9}$$

$$\Delta\,(I5/MCA) > 0 \tag{5.10}$$

$$\Delta\,(I5/VP3) < 0 \tag{5.11}$$

$$\Delta\,(I6/RF) > 0 \tag{5.12}$$

$$\Delta\,(I6/FhM) > 0 \tag{5.13}$$

$$\Delta\,(I6/VPA)\,\{false \rightarrow true\} > 0 \tag{5.14}$$

$$\Delta\,(I6/MCA) < 0\Delta\,(I6/VP1) < 0 \tag{5.15}$$

$$\Delta\,(I6/VP3) < 0 \tag{5.16}$$

$$\Delta\,(I7/VPA)\,\{false \rightarrow true\} < 0 \tag{5.17}$$

$$\Delta\,(I8/RfM)\,[0, enough] > 0 \tag{5.18}$$

$$\Delta\,(I8/VP2) > 0 \tag{5.19}$$

$$\Delta\,(I9/MCA) > 0 \tag{5.20}$$

$$\Delta\,(I9/VP3) > 0 \tag{5.21}$$

$$\Delta\,(I10/RF) < 0 \tag{5.22}$$

$$\Delta\,(I10/FhM) < 0 \tag{5.23}$$

$$\Delta\,(I10/VP1) > 0 \tag{5.24}$$

$$\Delta\,(I10/VP2) > 0 \tag{5.25}$$

$$\Delta\,(I10/VP3) > 0 \tag{5.26}$$

Table 5.3: Differential relations elicited for the Meeting Scheduler example [SLAM13]

AwReq	**Priority Value**
AR6	1.63
AR5	1.58
AR4	1.17
AR1	1.09
AR8	0.93
AR2	0.8
AR3	0.7
AR7	0.76
AR9	0.64
AR10	0.6

Table 5.4: Priority Values of AwReqs

AwReq	***EvoReq* operation**
AR1	Retry(50000ms)
AR2	Replace(AR2,AR2_200euro)
AR3	Replace(AR3,AR3_14d)
AR4	Retry(1day)
AR5	Retry(10000)
AR6	Replace(AR6,AR6_80%prt)
AR7	Warning()
AR8	Replace(AR8,AR8_14d)
AR9	Abort()
AR10	Replace(AR10,AR10_3days)

Table 5.5: Evoreq operations for AwReqs

change means that subsequent executions of the base system will see only R'. As it is presented in Table 5.5 $AR2$, $AR3$, $AR6$, $AR8$ and $AR10$ can be replaced by other requirements with weaker quality constraints. For example, for *Good participation*, instead of expecting 90% participation we could lower expectations to 80%. For the *AwReqs* $AR1$, $AR4$ and $AR5$ we do not weaken requirements but rather postpone dealing with them, using the Retry(time) operation. For AR7 and AR9 we use the *EvoReq* operations Warning() and Abort() respectively. The first one prints a warning message and the second one suspends the requirement altogether.

Now that all the required input for *Zanshin* has been specified we present a case of multiple failures and how these are resolved. The adaptation requirements for the framework are those presented in Figure 5.1 and the predefined value of $VP1$ is *Adapt Optimistically.* The monitor component checks periodically every 1 hour if there are any failures. In the first scenario the monitor detects failures of $AR1$, $AR2$. Then the Failure Manager collects the parameters than can tune failing indicators. According to Table 5.3 for $AR1$ the only option is to decrease RF (required fields to organize a meeting) and for $AR2$ either decrease RfM (Rooms for Meetings available) or decrease $VP2$. There are though priority conflicts with non-faillling *AwReqs* a) $AR1$ conflicts with $AR6$ and b) $AR2$ conflicts with $AR4$ and $AR8$. The Decision-Maker takes into account the adaptation goal *Adapt Optimistically* and ignores the priority conflicts. Therefore, the Strategy Executor composes an adaptation strategy that executes two operations $decrease(RF)$ and $decrease(RfM)$. The framework will wait for the effects to take place and then examines if the indicators still need improvement. The previous *AwReqs* are not failing anymore, but $AR4$ and $AR8$ are adversely affected as they are now failing. Moreover, the Adaptation Manager because of these new failures switches to *Adapt with Compensation.* The Decision Manager then decides to increase the param-

eter RfM to improve $AR4$ and $AR8$ and executes the assigned *EvoReq* operation for $AR2$ since it has the lowest priority and no other parameter to be reconfigured. The result is a new strategy with the operations $increase(RfM)$ and $Replace(AR2, AR2_200euro)$. The outcome is that the new $AR2$ is not failing anymore and $AR4$ and $AR8$ are not failing.

5.3.2 Improved Adaptation

The previous version of *Zanshin* and *Qualia* adaptation mechanism were ignoring the fact that there are cases where multiple failures cannot be handled independently by treating individually and sequentially indicators of failing *AwReqs*. The scenarios presented below demonstrate the advantages that the new adaptation mechanism offers.

Scenario 1: The monitor detects failures for $AR4$ and $AR8$.

***Qualia*:** The framework will treat first the failure that was detected first, in this case $AR4$. The parameter RfM is increased due to the differential relation (7) of Table 5.3. If after the change $AR4$ is not failing a new adaptation session starts for $AR8$ increasing $VP2$ due to the differential relation (18). When $AR8$ ceases to fail as a consequence of the parameters increment $AR2$ fails because of the negative impact the changes (differential relations (4) and (5)). Then *Qualia* would decrease again either RfM or $VP2$ (or both) in order for $AR2$ to recover. It is obvious that such an adaptation mechanism will fall into an infinite loop doing and undoing the same changes.

***Qualia+*:** The framework is instructed how to adapt in these cases using *AdReqs* as described in the example of the previous subsection.

Scenario 2: The monitor detects failures for $AR5$, $AR6$ and $AR9$.

***Qualia*:** The framework treats again the failures sequentially changing parameters that would result again in an infinite loop.

***Qualia+*:** The Failure Manager finds MCA (maximum conflicts allowed among the participants' time-schedules) as only available parameter which means that the Decision Maker has to choose between increasing MCA to improve $AR5$ and $AR9$ and worsen $AR6$ or decrease MCA to improve $AR6$ and worsen $AR5$ and $AR9$. The choice is based on the priority values of the indicators shown in Table 5.4 and since $(1.58 + 0.64 > 1.63)$ AR5 and AR9 are preferred. The finally result would be a strategy with the operations: a) increase(MCA) and b) $Replace(AR6, AR6_80\%prt)$. This way we compensate for not improving *AwReq* while meeting adaptation requirements.

We note that adaptation strategies could have also been composed defining rules that specify in what order should *EvoReqs* operations be executed and under what circumstances. However, such rules are outside the scope of requirements and hard for stakeholders to conceptualize, and therefore define. *Qualia+* allows stakeholders to define the policies by which such conflicts should be resolved during the adaptation process. Moreover, the dynamic composition of adaptation strategies means that the adaptation process does not need to go off-line when adaptation requirements are changed.

5.4 Chapter Summary

In this chapter we presented a requirements-based adaptation mechanism that is able handle multiple concurrent requirements failures. To accomplish this, we have extended the *Zanshin* framework with two basic new features. The first is the concept of *AdReqs*, which are requirements about the adaptation process itself. Like all requirements, these come from the

stakeholders and define policies the adaptation process has to comply with. The second feature is a decision making mechanism that takes into account *AdReqs* to decide which requirements should be improved and which have to be compensated temporarily or permanently by *EvoReqs* operations. The new adaptation mechanism, called *Qualia+*, makes it possible for stakeholders to reflect their needs and preferences for the adaptation process by assigning priorities and compensation operations to base system requirements. Moreover, the fact that adaptation strategies are composed dynamically allows stakeholders to change *AdReqs* during system operation.

Chapter 6

The Next Adaptation Problem

> The formulation of the problem is often more essential than its solution, which may be merely a matter of mathematical or experimental skill.
>
> Albert Einstein

In this chapter we describe how the problem of choosing values for control parameters when indicators fail can be formulated as a constrained multi-objective optimization problem. More specifically, we focus on the *Next Adaptation Problem*, concerned with the selection of a new adaptation to address one or more failures. One of the main challenges for any adaptation mechanism is to select an optimal adaptation relative to one or more objective functions, such as minimizing cost of adaptation, minimizing degree of failure, and/or maximizing customer value. Hereby, we answer the **RQ4:** *How the self-adaptation problem is formulated as an optimization problem and how it can be solved?*

We answer this question by proposing a framework that does not just choose a good adaptation for the failing requirements, but actually selects an optimal one, relative to user-specified objective functions. In particular, given an analytical model that describes the relation between requirements success rates and control parameters, and given a set of failing require-

ments, the adaptation mechanism searches for new values for control parameters that reduce the degree of failure, while optimizing given objective functions which are related to quality attributes of the system. A quality attribute is a measurable quantity associated with the goals of the system e.g. time required to schedule a meeting or the average cost of meetings. If there are several quality attributes, the adaptation chosen optimizes them lexicographically [Ise82], i.e. best adaptations are selected relative to the most important objective function, among those best adaptations are selected relative to the second most important objective function, etc. Finally, we evaluate our approach with two exemplars, namely the Meeting Scheduler and an e-shop.

6.1 Problem Formulation

Tuning a control parameter results in a change of value of certain indicators and quality attributes. Therefore, after every diagnosis, the available goals and tasks are annotated with the potential contributions that can provide to the associated indicators as shown in Figure 6.1. In this example the full form of the goal model (the possible values for CVs are represented as OR-refinements) is captured along with impact of each goal or task to indicators and quality attributes based on the *control parameter profile* presented in Table 6.1. More specifically, increasing FhM from 70% to 100% will result in a decreased $I3$ by 1.5%, whereas the increasing the percentage of maximum allowed conflicts (MCA) over the number of invitees from 20% to 40% increases $I3$ by 1.6%. Each alternative results in different time that it takes for the goal *Find Date* to be fulfilled and therefore there is an annotation with the value of the quality attribute $find_date_time$ for each of them. The values of Table 6.1 as well as the those for quality attributes are provided by domain experts and can be updated if necessary to increase

precision.

Table 6.1: Control Parameter Profile.

ΔCP	$\Delta I3(\%)$
MCA	± 0.08
FhM	∓ 0.05
$VP3\{automatically \rightarrow manually\}$	$+6$

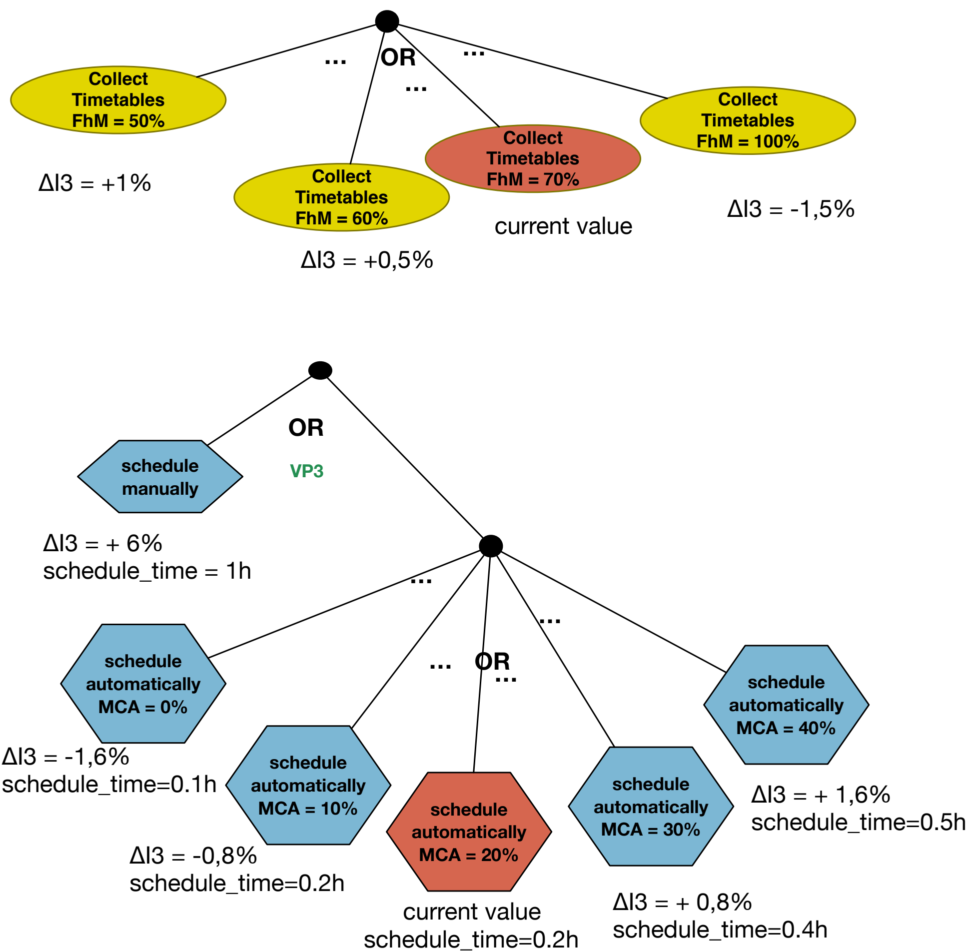


Figure 6.1: Goal model annotated with contributions

Each time one or more indicators fail, the goal model must be annotated based on what were the previous values of the control parameters and the control parameter profile.

When the measured value of an indicator I_j^m is below the threshold I_j^o by $R_j = I_j^o - I_j^m$, a new adaptation is triggered in order to minimize R_j (ideally, it should be zero) for every indicator.

Definition 1 (Indicator Cost-Function) *Let $D_I = \{I^m, I^o, R, AS'\}$, be the diagnosis for the indicator I, where I^m is its measured value, I^o is its threshold, $R = I^o - I^m$ and AS' the set of all available goals or tasks that can contribute to the current value of I positively, negatively or zero. An Indicator Cost-Function F^I is defined as $F^I = R + \sum \Delta I$, where $\sum \Delta I$ is the the sum of contributions that I will receive by the next adaptation.*

According to Figure 6.1 if the next adaptation includes $FhM = 60\%$ and *schedule manually* is selected, $\sum \Delta I3 = 0.5+6 = 6.5\%$. Therefore, the indicator $I3$ is going to be increased by 6.5%. The target of the adaptation mechanism is to minimize all Indicator Cost-Functions. However, due to the presence of conflicting contributions among the indicators the adaptation mechanism needs to settle for a trade-off. Towards this direction, we prioritize all indicators using AHP, eliciting weights that represent their importance.

Definition 2 (Global Cost-Function) *Let F be the set of all Indicator Cost-Functions and W the set of their respective weights. A Global Cost-Function F^G is defined as $F^G = \sum w_j \times F_j^I$, where $w_j \in W$ and $F_j^I \in F$.*

The role of the adaptation mechanism is twofold. First, a configuration of the goal model must be found so that the root goal is satisfied while the Global Cost-Function is minimized. In other words, the next adaptation problem consists of a combination of two different problems a) satisfiability of all constraints and b) multi-objective optimization. Such combined problems are solved by reasoning technologies, notable Satisfiability and Optimization Modulo Theories (SMT/OMT) [ST15].

In Figure 6.1 apart from the contributions to the indicators, goals and tasks are also annotated with certain kinds of costs. For example, the value selection for MCA includes the amount of time it takes to schedule a meeting's date (finddate_time), whereas the value for FhM determines the time it takes to collect timetables ($collection_time = FhM \times 0.02$). Stakeholders, usually require the satisfaction of their goals with the minimal cost adaptation. This means that the total time for scheduling a meeting ($total_scheduling_time = collection_time + find_date_time$) must also be minimized. The *Next Adaptation Problem* can encompass optimizations relative to other costs, such as $total_scheduling_time$.

Definition 3 (Next Adaptation Problem) *Let $P = \{F^G, QA_1, .., QA_n, AS'\}$ be a tupple where F^G is a global cost-function, $QA_1, ..., QA_n$ a set of quality attributes, and AS' the new adaptation space that includes all the available control parameters. The Next Adaptation Problem refers to finding an optimal configuration over the goal model that minimizes F^G and lexicographically optimizes each quality attribute (based on stakeholder preferences), wrt to the availability of goals and tasks in AS'.*

6.2 Prometheus Framework

In the previous section we presented how the adaptation process is modelled as an optimization problem using goal models and a quantitative information about the relationship between control parameters and indicators. This section describes the steps of the adaptation process at runtime as well as the proposed Prometheus framework, whose architecture is shown in Figure 6.2.

Prometheus interacts with the target system and its environment through monitors and actuators that are the responsibility of system designers to build and usually are application-specific. The internal mechanism of

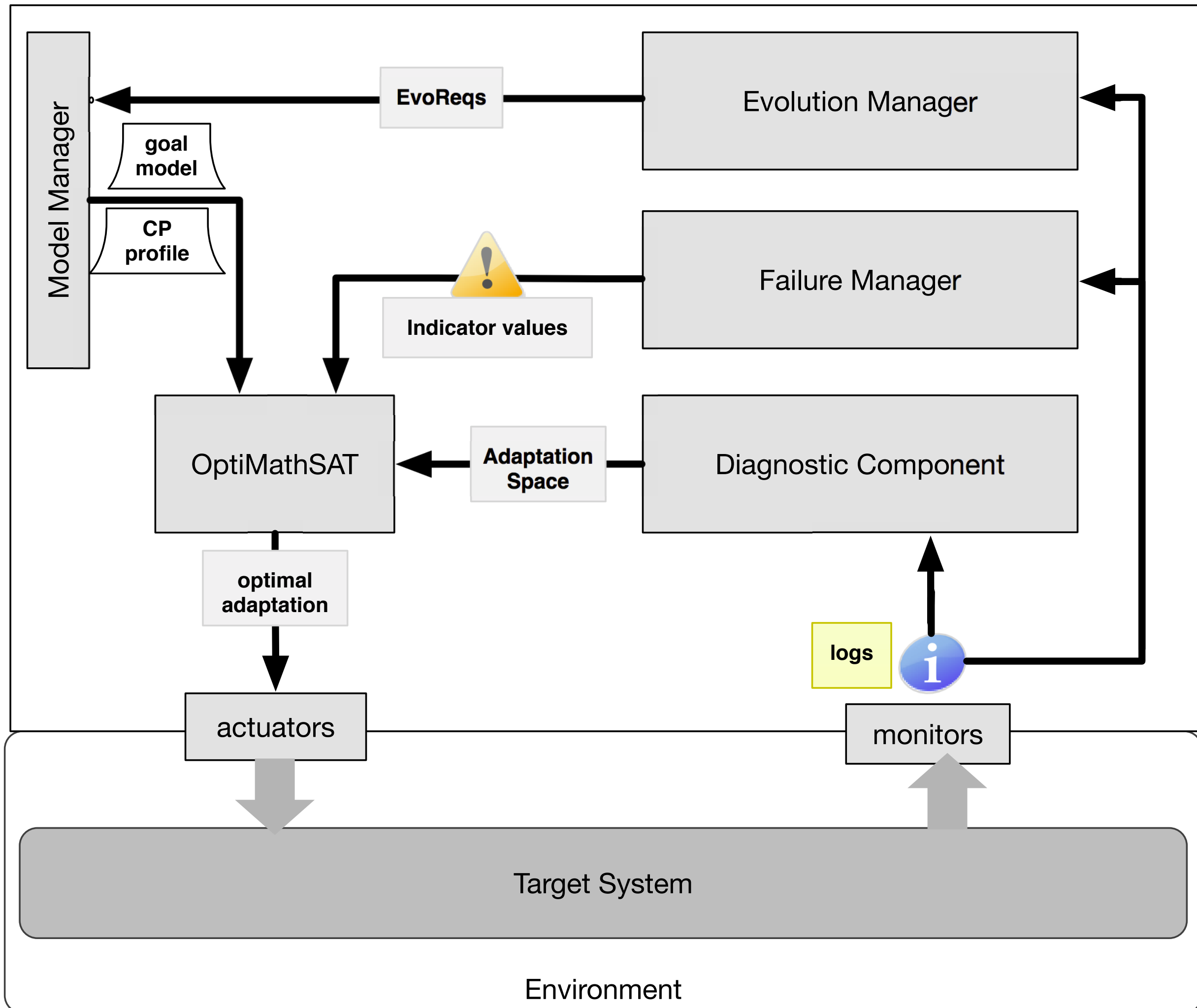


Figure 6.2: Prometheus framework

Prometheus consists of five components described below.

Diagnostic Component This component reads the system logs and reasons about the root causes of the identified failures. More specifically, discovers denied domain assumptions or failing tasks that could not be performed. These domain assumptions and tasks are marked as *denied* in the new available adaptation space constraining the available options for finding the optimal next adaptation. For instance, if the domain assumption $DA1$ in Figure 6.3 is denied then $t3$ cannot be selected as an option for

collecting timetables and due to a goal constrain neither $t8$ can be selected for finding a date.

Failure Manager This component reads the system logs and measures the success rates of each indicator. When the measured value of one or more indicators is found below the threshold imposed by the associated *AwReq* a new adaptation is triggered and a new configuration over the goal model must be chosen.

Evolution Manager This component reads system logs and checks if any precondition holds; if it does, the goal model is updated in accordance with the *EvoReq*.

Model Manager This component stores the control parameter profile of the system and the elicited goal model. Each time a new adaptation is triggered the goal model is annotated with the impact values of each goal to the indicators.

OptiMathSAT This component receives the annotated goal model along with the new adaptation space that disables a certain amount of alternatives. Based on this model and the measured values of the indicators produces an optimal next adaptation.

To give a better understanding of how the framework operates, we describe every step followed for finding the next adaptation. The steps are:

1. **Collect system logs.** The success or failure of the executed tasks is recorded in logs collected by the designed monitors.

2. **Detect failures.** The Failure Manager compares measured values of the indicators with those that are imposed by their associated *AwReqs*. If one or more failures are detected a new adaptation is triggered.

3. **Find root causes.** The diagnostic tool provides the new adaptation space which excludes the goals that caused the detected failures and marks as denied the domain assumptions that do not hold any more.

4. **Apply *EvoReqs*.**The Evolution Manager updates the goal model with *EvoReqs* if any of the preconditions specified applies.

5. **Annotate the goal model.** The Model Manager annotates the goal model based on the control parameter profile.

6. **Find Optimal Next Adaptation.** OptiMathSAT produces an optimal adaptation.

7. **Apply new adaptation.** The new adaptation is applied to the target system by the actuators.

6.3 Evaluation

In this section we describe the Meeting-Scheduler and E-shop exemplars as well as the evaluation process of *Prometheus* by using failure scenarios.

6.3.1 The Meeting-Scheduler Exemplar

As we discussed in the previous chapters, the main goal of the Meeting-Scheduler application is to receive meeting requests and produce meeting bookings. Figure 6.3 captures system goals. For the top goal to be satisfied, timetables must be collected (satisfy $G1$), make a booking for every meeting (satisfy $G2$) and allow the meeting organizers to manage their meetings (satisfy $G3$). The timetables can be collected by phone, by e-mail, or automatically by the system. However, the third option is available only if the meeting participants are using the system calendar. Next, booking a meeting involves finding a location and an appropriate date. The system

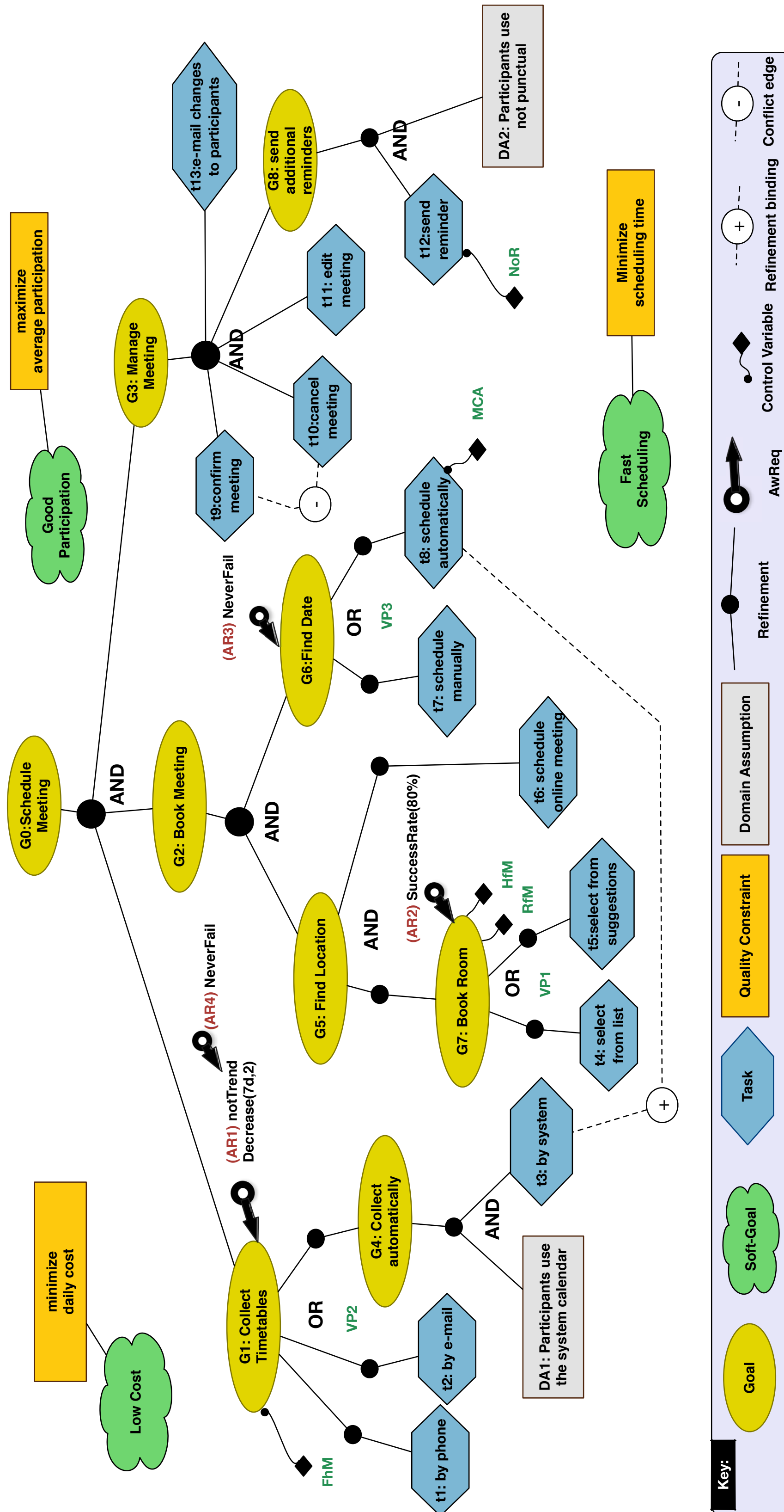


Figure 6.3: Meeting-Scheduler goal model

offers at the same time the opportunity to book a room or schedule an online meeting in case rooms are not available. A room can be selected from the entire list of existing rooms or from the suggested rooms by the system ($t4$ and $t5$ respectively). In the same manner, a date can be selected manually by the meeting organizer or the system selects one automatically ($t7$ and $t8$ respectively). A date though can be selected automatically only if the timetables are collected automatically and vice versa. Finally, meeting organizers can confirm, cancel and edit their meetings ($t9$, $t10$ and $t11$ respectively), while the system is responsible for e-mailing the participants in case a modification takes place ($t13$) and send more reminders in case the participants are not punctual enough.

To monitor the success of these requirements, four *AwReqs* are placed over the certain goals that are prone to failure. $AR1$ dictates that goal $G1$ must not fail more than twice a week and $AR4$ imposes on $AR1$ to never fail. Next, $AR2$ prescribes that at least for 80% of the meetings a room must have successfully be found and $AR3$ that $G6$ must always be fulfilled.

In addition to FhM and MCA, other control parameters are available to manipulate the success rate of the indicators associated to the aforementioned *AwReqs*. First, the goal *Book Room* is related with two CVs that control the number of local rooms RfM and hotel rooms HfM that are reserved for meetings. The number of additional reminders NoR associated to task $t12$ is yet another CV. Along with the CVs there are three VPs that stem from the OR-refinements of the goal model. In Table 6.2 the full control parameter profile for the Meeting-Scheduler application is presented.

In Figure 6.3 three soft goals are specified to capture the non-functional requirements of the system. The *Low Cost* soft goal is associated with the quality attribute *daily_cost*. The cost of a hotel rooms is 20€ and the daily cost for calls is *call_cost* = 30€ if timetables are collected by phone,

Table 6.2: Control Parameter Profile.

ΔCP	$\Delta I2(\%)$	$\Delta I3(\%)$	$\Delta I4(\%)$
MCA	0	± 0.08	0
FhM	0	∓ 0.05	0
RfM	± 2.1	0	0
HfM	± 2.7	0	0
NoR	0	0	0
$VP1\{t4 \rightarrow t5\}$	+2	0	0
$VP1\{t5 \rightarrow t4\}$	−2	0	0
$VP2\{t1 \rightarrow t2\}$	0	0	−2
$VP2\{t1 \rightarrow G4\}$	0	0	+6
$VP2\{t2 \rightarrow t1\}$	0	0	+4
$VP2\{t2 \rightarrow G4\}$	0	0	+6
$VP2\{G4 \rightarrow t1\}$(DA1 is true)	0	0	−5
$VP2\{G4 \rightarrow t2\}$(DA1 is true)	0	0	−6
$VP2\{G4 \rightarrow t1\}$(DA1 is false)	0	0	+5
$VP2\{G4 \rightarrow t2\}$(DA1 is false)	0	0	+3
$VP3\{t7 \rightarrow t8\}$	0	−2	0
$VP3\{t8 \rightarrow t7\}$	0	+6	0

otherwise $call_cost = 0$€. Therefore $daily_cost = 20 \times HfM + call_cost$ is a quality attribute that must be minimized. Next, the soft goal *High Participation* is associated with the quality attribute $average_participation = 80 + 0.2 \times FhM - 0.2 \times MCA + 5 \times NoR(\%)$ which must be maximized. When $VP3 = t7$ then $MCA = 20$. Finally, the soft goal *Fast Scheduling* is calculated as described in the previous sections.

To evaluate our approach we implemented a simulation of the Meeting-Scheduler application. In this simulation we encoded a failure scenario and inserted it as input to both *Prometheus* and *Zanshin* integrated with the *Qualia* as described in [SLM12a]. Both frameworks are requirements-based and this has been our main motivation for carrying out this comparison.

```
! I2 = 72%, I3 = 94%, I4 = 87%
Current Configuration:
VP1=t5, VP2=t3, VP3=t8
MCA=10, FhM=70, NoR=0, RfM=6, HfM=4
! DA1 = false  DA2 = false
! No EvoReqs apply
P1: MCA=20, FhM=78, NoR=0, RfM=12, HfM=3
    VP1=t4, VP2=t1, VP3=t7
P2: MCA=20, FhM=78, NoR=0, RfM=3, HfM=10
    VP1=t5, VP2=t1, VP3=t7
----------------------------------
Output
P1: I2=79.9%, I3=100%, I4=92%
    total_cost = 90
    average_participation = 91.6%
    total_scheduling_time = 6.6hrs
P2: I2=78%, I3=100%, I4=92%
    total_cost = 230
    average_participation = 91.6%
    total_scheduling_time = 6.6hrs
```

Listing 6.1: *Prometheus* Output

However, *Zanshin* uses qualitative information for capturing the impact of control parameters on indicators and does not optimize as opposed to the quantitative and optimizing approach of *Prometheus*. Moreover, the default adaptation algorithm of *Zanshin* select randomly a control parameter that contributes positively to the treated failure, which is increased by a predefined amount of units. In order also to demonstrate the importance of lexicographic optimization we execute the adaptation process of *Prometheus* with and without optimizing quality attributes. Among the quality attributes cost is optimized first, participation second and scheduling time third. In Listing 6.1 the results of the simulation are presented. $P2$ marks the next adaptation and the consequent output of *Prometheus* when only the Global Cost-Function is optimized whereas $P1$ also optimizes (lexicographically) with respect to quality attributes. Finally, Z in Listing 6.2 indicates the next adaptation and output produced by *Zanshin*.

The results in Listing 6.1 and Listing 6.2 show that *Prometheus* per-

```
! I2 = 72%, I3 = 94%, I4 = 87%
Current Configuration:
VP1=t5, VP2=t3, VP3=t8
MCA=10, FhM=70, NoR=0, RfM=6, HfM=4
! DA1 = false  DA2 = false
! No EvoReqs apply

----------------------------------
Output
Iteration 1
Z:  I2=72.54%, I3=100%, I4=92%
Z:  MCA= 20, FhM=70, NoR=0, RfM=6, HfM=6
    VP1=t5, VP2=t1, VP3=t7
    total_cost = 150
    average_participation = 90%
    total_scheduling_time = 6.6hrs
Iteration 2
Z:  I2=73.58%, I3=100%, I4=92%
Z:  MCA= 20, FhM=70, NoR=0, RfM=6, HfM=8
    VP1=t5, VP2=t1, VP3=t7
    total_cost = 190
    average_participation = 90%
    total_scheduling_time = 6.6hrs
Iteration 3
Z:  I2=74.46%, I3=100%, I4=92%
Z:  MCA= 20, FhM=70, NoR=0, RfM=10, HfM=8
    VP1=t5, VP2=t1, VP3=t7
    total_cost = 190
    average_participation = 90%
    total_scheduling_time = 6.6hrs
Iteration 4
Z:  I2=75.84%, I3=100%, I4=92%
Z:  MCA= 20, FhM=70, NoR=0, RfM=14, HfM=8
    VP1=t5, VP2=t1, VP3=t7
    total_cost = 190
    average_participation = 90%
    total_scheduling_time = 6.6hrs
Iteration 5
Z:  I2=79.44%, I3=100%, I4=92%
Z:  MCA= 20, FhM=70, NoR=0, RfM=18, HfM=10
    VP1=t5, VP2=t1, VP3=t7
    total_cost = 230
    average_participation = 90%
    total_scheduling_time = 6.6hrs
Iteration 6
Z:  I2=82.5%, I3=100%, I4=92%
Z:  MCA= 20, FhM=70, NoR=0, RfM=22, HfM=10
    VP1=t5, VP2=t1, VP3=t7
    total_cost = 230
    average_participation = 90%
    total_scheduling_time = 6.6hrs
```

Listing 6.2: *Zanshin* Output

forms better than *Zanshin* since it exploits quantitative relations between control parameters and indicators. On the other hand, *Zanshin* changes by a fixed amount randomly chosen control parameters that is known to improve the failing indicators. This means that it would take more iterations for *Zanshin* to fix a failure or to minimize it. More specifically, in the simulated failure scenario it took six steps until $I3$ and $I4$ converges to the best possible value whereas $I2$ slightly overshoots. Moreover, the results indicate that the qualities of the system are not taken into account while deciding the next adaptation. Large number of iterations for the indicators to converge to their prescribed thresholds, implies that *Zanshin* is not suitable for rapidly changing environment, since by the tame a good solution is found the failing indicators might be different. Given the result of the simulation lexicographic optimization can provide the same results for the failing indicators as the optimization of the Global Cost-Function only, but also considerably improves quality attributes, such as *total_cost* in this case.

6.3.2 The E-shop Exemplar

The main goal of the E-shop application is to place orders of goods that clients buy online. Figure 6.4 captures the goals for this system. For the top goal to be satisfied the customer must select the product they would like to order (goal $G1$) and check-out the order (goal $G2$). The customer is able to search for products they are interested in (task $t1$), view the details of the product (goal $G5$) in textual mode (task $t3$) or multimedia mode (task $t4$). For an order to be checked out the customer must fist login (task $t5$). Then, the product's availability is checked (goal $G3$). The goal $G3$ can be fulfilled either by making a new order (goal $G6$) or by removing the selected item from the stock list (taksk $t9$). The product is ordered either from a retailer (goal $G7$) or from a wholesaler (goal $G8$). A precondition for

sending an order to a retailer or a wholesaler (tasks $t7$ and $t8$ respectively) is a retailer or a wholesaler to be available ($DA1$ and $DA2$ respectively).

For the requirements of this exemplar we elicited three *AwReqs*. $AR1$ prescribes that multimedia mode for viewing product details must by used 10 times more than textual model and according to $AR2$ this constraint must not fail more than 80% of the times. Next, the goal $G2$ must not fail more than 95%.

The elicited *AwReqs* are related to three control parameters. $AR2$ can be controlled by changing the value of $VP1$, or in other words witching textual to multimedia mode and vice versa and the number of servers (NoS) that are deployed to host the webpage of the e-shop. Finally, $VP1$ and $VP2$ can be used to control the success of $AR3$.

Table 6.3: Control Parameter Profile.

ΔCP	$\Delta I2$(%)	$\Delta I3$(%)	response time(ms)	cancellation rate(%)
NoS	± 1.2	0	∓ 200	0
$VP1\{t3 \rightarrow t4\}$	3	0	+1000	0
$VP1\{t4 \rightarrow t4\}$	3	0	+200	0
$VP1\{t4 \rightarrow t3\}$	-1	0	+1	0
$VP1\{t3 \rightarrow t3\}$	-0.2	0	0	0
$VP2\{G6 \rightarrow t9\}$	0	+3.4	0	+4
$VP2\{t9 \rightarrow G6\}$	0	0	0	-4
$VP3\{G7 \rightarrow G8\}$ (DA1 is false)	0	+1.2	0	0
$VP3\{G8 \rightarrow G7\}$ (DA2 is false)	0	+0.8	0	0

In Figure 6.4 four soft goals are specified to capture non-functional requirements for the e-shop exemplar. First, *High Performance* is associated with the quality attribute *response_time* of the deployed servers, which in this case is not only minimized but also constrained to be lower

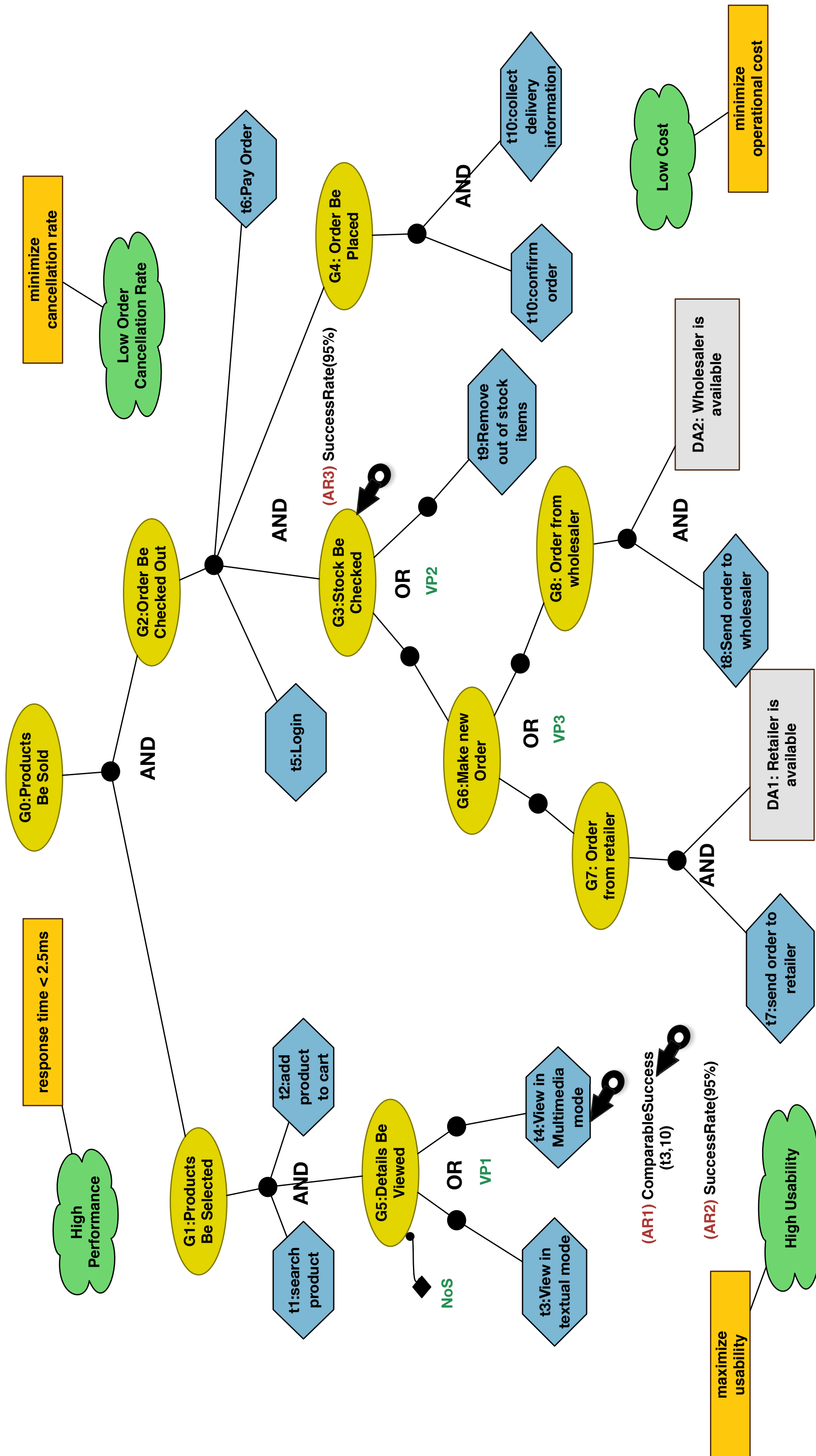


Figure 6.4: E-shop goal model

```
! I2 = 93%, I3 = 94%
!response time = 4000ms
!cancellation rate = 10 %
Current Configuration:
VP1=t4, VP2=G6, VP3=G7
NoS=3
! DA1 = false  DA2 = true
! No EvoReqs apply

P:  NoS=4
    VP1=t4, VP2=G6, VP3=t8
  ---------------------------------
Output
P:  I2=99.6%, I3=95.2%
    usability = 5
    response time = 1400ms
    cancellationRate =6 %
    cost = 1920 euro
```

Listing 6.3: *Prometheus* Output

than 2.5s. The disposal of more servers results in lower response time. Next, *High Usability* is associated to the quality attribute *usability* which taakes the value 5 when the website is viewed in multimedia mode and the value 3 when text mode is selected. Next, *Low Cost* relates to the quality attribute $operational_cost = server_cost + ordering_cost$, where $server_cos = 120 \times NoS$ and $ordering_cost$ is the cost of making a new order which is 1200€ in case the products are ordered from a wholesaler and 1000€ in case it ordered from a retailer. The prices might vary through time since they depend on which products are mostly sold in a particular period of time and in what quantities. Therefore, it is responsibility of the domain experts to update this numbers. Finally, *Low Order Cancellation Rate* is associated with the quality attribute *cancellation_rate*.

As for the Meeting-Scheduler exemplar also in this case we simulated a failure scenario and we compare the responses of *Prometheus* and *Zanshin*. The control parameter profile is presented in Table 6.3 whereas the results output of *Prometheus* and *Zanshin* are depicted in Listing 6.3 and Listing 6.4 respectively.

```
! I2 = 93%, I3 = 94%
!response time = 4000ms
!cancellation rate = 10 %
Current Configuration:
VP1=t4, VP2=G6, VP3=G7
NoS=3
! DA1 = false  DA2 = true
! No EvoReqs apply
  ----------------------------------
Output
Iteration 1
Z:  NoS=4
      VP1=t4, VP2=t9, VP3= -
Z:  I2=97.2%, I3=97.4%
    usability = 5
    response time = 4000ms
    cancellationRate = 14 %
    cost = 480 euro
```

Listing 6.4: *Zanshin* Output

The results in Listing 6.3 and Listing 6.4 show that both frameworks managed to find a good solution for the failing indicators. However, as in the previous case *Prometheus* managed to find optimal solutions for the soft goals as well.

6.3.3 Discussion

The experiments ran on a computer with an Intel i5 processor at 2.5GHz and 16GB of RAM. Both *OptiMathSAT* [ST15] and its extension *CGM−tool* [NSGM16b] which has been the basis of the *Prometheus* prototype have been tested in terms of scalability and can handle up to 8000 nodes in negligible time. Compared to *Zanshin*, *Prometheus* is able to select among all the equivalent solutions the one that optimizes the quality attributes of the system as well.

The main bottleneck of the approach is producing the control parameter profile, which is a human-driven process and the time overhead depends on the amount of the control parameters and the level of expertise of the software designers.

6.4 Chapter Summary

In this chapter we proposed a framework that can compose an optimal adaptation when one or more requirements fail. The optimal nature of the produced adaptation refers to the minimization of the degree of failure of the system's functional requirements while non-functional properties of the system are optimized lexicographically according to stakeholder priorities.

Our proposed framework is built on top of a diagnostic component that reads the logs of the monitored system and reasons about the causes of failure. Failing domain assumptions and tasks define the new adaptation space where all the potential solutions lie in. Then, each alternative of the adaptation space is annotated with the quantitative impact that will bare to the indicators. Finally, the *OptiMathSAT* solver finds the best alternative in the new adaptation space.

The contribution of the chapter of this thesis, is a requirements-driven approach that determines optimal adaptations for multiple failures and with respect to multiple objective functions. Moreover, we have demonstrated experimentally that our proposal performs better than a qualitative, requirements-driven framework (*Zanshin*), and also established that our proposal works for real world-size problems.

Chapter 7

Control-based design of self-adaptive software

All models are wrong; some models are useful.

George E. P. Box

In Chapter 1 we mentioned that one the fundamental components of self-adaptive systems, the feedback loop, is adopted from Control Theory. Then, in Chapter 2 we presented various approaches that have as a basis a controller that decides the values of control parameters in order to reduce the control error of the monitored goal(s). Using a controller as an adaptation mechanism provides also a set of formal guarantees that we described in Chapter 1 as SASO properties.

A key element of software adaptation is to capture the dynamics of the controlled system. In the previous chapter the adopted model ignored the time dimension in the relation between control input and control output. More specifically, we assumed that the expected increase/decrease of an indicator after changing the value of one or more control parameters will be instant. However, this assumption does not always hold and more advanced models are required, that involve also the time dimension. In Section 2.1 we presented the analytical model, shown below, for capturing also the

impact of time on the composition of adaptation strategies.

$$\begin{cases} x(t+1) = Ax(t) + Bu(t) \\ y(t) = Cx(t), \end{cases} \tag{7.1}$$

In this chapter we present a control-theoretic approach, named Model Predictive Control that has been used to solve myriad of problems in other engineering disciplines, such as automotive engineering, chemical engineering etc. More specifically, we illustrate step by step how an MPC controller is designed and how each of its components relates to the software engineering ones we have used until now for developing adaptation mechanisms. Our purpose is to put the foundations for building an adaptation framework that includes an MPC controller as a decision mechanism. This framework is described in detail in the next chapter.

7.1 Model Predictive Control

Based on the dynamic model of Equation (7.1), different control strategies can be designed. Hereby, we present a *receding horizon* MPC [CBA04, Mac02] that is able to manage the achievement of multiple conflicting goals by means of multiple control parameters. When the controller is complemented with a Kalman Filter (KF) [Lju99], it can learn online how to adapt the controller to the system's behaviour, overcoming inevitable inaccuracies coming from dynamics not captured from model (7.1) and unknown disturbances acting on the system.

MPC is a control technique that formulates an optimization problem to use a set $u(\cdot)$ of control parameters (actuators) to make a set of indicators $y(\cdot)$ achieve a set of goals $y^{\circ}(\cdot)$ over a prediction horizon H. At every control instant t, the values of the control parameters $u^{\star}$ are obtained by minimizing a cost function J_t, subject to given constraints. The optimiza-

tion problem includes a prediction of the future behaviour of the system based on the dynamic model (7.1). An obtained solution is therefore a plan of the future control parameter values $u^\star = \left[u_t^\star, u_{t+1}^\star, \ldots, u_{t+H-1}^\star\right]$ over the prediction horizon. This planning is especially needed in the case of delay in the effects of changes of control parameters. For example, increasing the number of hotel rooms requires approval by the administration council that meets only every 2 days. Hence, the adaptation mechanism must be aware of when changes to control parameters impact on the indicators and make look-ahead plans. According to the receding horizon principle, only the first computed value $u_t^\star$ is applied to the system, i.e. $u(t) = u_t^\star$. The reason is that for real-world systems, it is impossible to derive perfect models that describe their dynamic behaviour. Therefore, the plan must be corrected at each step and the horizon recedes by one unit. Another reason the plan might fail is a change in the external disturbances (e.g. system workload). In other words, the plan would have been followed as is only if a perfect model were available and no disturbances were present, which in practice is impossible. To tackle this obstacle, at the next control instant, a new plan is computed according to the new measured values of the indicators. This accounts for modelling uncertainties, and possible unpredictable behaviours of the system that are not captured by model (7.1).

7.1.1 Formal description

In order to present the underlying rationale of the MPC, it is convenient to rewrite dynamic model (7.1) in an "augmented velocity form":

$$\overbrace{\begin{bmatrix} \Delta x(t+1) \\ y(t) \end{bmatrix}}^{\tilde{x}(t+1)} = \overbrace{\begin{bmatrix} A & 0_{n\times p} \\ C & I_{p\times p} \end{bmatrix}}^{\tilde{A}} \overbrace{\begin{bmatrix} \Delta x(t) \\ y(t-1) \end{bmatrix}}^{\tilde{x}(t)} + \overbrace{\begin{bmatrix} B \\ 0_{p\times m} \end{bmatrix}}^{\tilde{B}} \Delta u(t)$$

$$y(t) = \overbrace{\begin{bmatrix} C & I_{p\times p} \end{bmatrix}}^{\tilde{C}} \overbrace{\begin{bmatrix} \Delta x(t) \\ y(t-1) \end{bmatrix}}^{\tilde{x}(t)} \tag{7.2}$$

Here, $\Delta x(t) = x(t) - x(t-1)$ is the state variation and $\Delta u(t) = u(t) - u(t-1)$ is the control increment. The output of the system $y(t)$ is unchanged, but is now expressed with respect to the state variations $\Delta x(t)$ and not with respect to the state values $x(t)$. The new dynamic model (7.2) is used as a prediction model over a finite horizon H. This means that the controller will use it to predict what values of the states and of the indicators are going to be after H time steps from the current one. The MPC controller minimizes the cost function

$$J_t = \sum_{i=1}^{H} \left[y^{\circ}_{t+i} - y_{t+i}\right]^T Q_i \left[y^{\circ}_{t+i} - y_{t+i}\right] \tag{7.3}$$

$$+ \left[\Delta u_{t+i-1}\right]^T P_i \left[\Delta u_{t+i-1}\right], \tag{7.4}$$

where $Q_i \in \mathbb{R}^{p\times p}$ and $P_i \in \mathbb{R}^{m\times m}$ are symmetric positive semi-definite weighting matrices, that respectively represent the importance of the distance between the goals and the current values and the "inertia" in changing the values of the actuators. In particular, Q_i is a diagonal matrix that contains the values of the set of weights that can be obtained by applying Analytical Hierarchy Process (AHP) [KR97], in which the stakeholders

perform pairwise comparisons to prioritize the elicited goals. This means that when not all the goals are simultaneously feasible (for example because one conflicts with another), the controller will favour the satisfaction of the goals with the higher weights. The matrix P_i preferences over control parameters. When a control parameter is requested not to change frequently its value the assigned weight must be relatively smaller than most of the weights of the other control parameters. In the following we will consider the weight matrix Q as $Q := Q_1 = Q_2 = \ldots = Q_H$, and the weight matrix P as $P := P_1 = P_2 = \ldots = P_H$, i.e., the weight matrices are considered to be constant along the prediction horizon.

The resulting MPC optimization problem can written as follows:

$$\begin{aligned} \text{minimize}_{\Delta u_{t+i-1}} \quad & J_t \\ \text{subject to} \quad & u_{\min} \leq u_{t+i-1} \leq u_{\max}, \\ & \Delta u_{\min} \leq \Delta u_{t+i-1} \leq \Delta u_{\max}, \\ & \tilde{x}_{t+i} = \tilde{A} \cdot \tilde{x}_{t+i-1} + \tilde{B} \cdot \Delta u_{t+i-1}, \\ & y_{t+i-1} = \tilde{C} \cdot \tilde{x}_{t+i-1}, \\ & \quad i = 1, \ldots, H, \\ & x_t = x(t). \end{aligned} \tag{7.5}$$

This formulation is equivalent to a convex Quadratic Programming (QP) problem [Mac02]. The problem has time complexity $\mathcal{O}(H^3 m^3)$ [WB10]. A solution to the problem consists of a plan of optimal future $\Delta u^{\star}_{t+i-1}$, $i = 1, \ldots, H$, but only the first one is applied, i.e., $\Delta u(t) = \Delta u^{\star}_t$, as we explained earlier. The new control signal is then:

$$u(t) = u(t-1) + \Delta u(t). \tag{7.6}$$

The MPC strategy assumes that the state of the system is measurable, but in many cases this is not possible. Indeed, since there is often no correlation with physical quantities, it is impossible to give a meaningful

interpretation to $x(t)$, hence it is impossible to measure. However, based on the dynamic model (7.1), it is possible to estimate its value measuring the values of $y(t)$ and $u(t)$. To accomplish this, we here use a KF that finds an estimate $\hat{x}(t+1)$ of the state $x(t+1)$, measuring the applied control signal $u(t)$ and the output $y(t)$.

$$\begin{aligned} \hat{y}(t) &= C\hat{x}(t) \\ \hat{x}(t+1) &= A\hat{x}(t) + Bu(t) + K\left(y(t) - \hat{y}(t)\right) \end{aligned} \tag{7.7}$$

Note that the variables of the KF are commonly denoted by a “hat”, i.e., $\hat{x}(k)$ and $\hat{y}(k)$, to distinguish them from the variables of the dynamic model (7.1). Based on the state estimate $\hat{x}(t)$, the KF shown in (7.7) computes an estimate of the output $\hat{y}(t)$, to measure the difference between the predicted value $\hat{y}(t)$ and the real value $y(t)$. The value of K, called *Kalman gain*, weights the discrepancy between the predicted value $\hat{y}(t)$ and the real value $y(t)$, adjusting the dynamics of the KF [Lju99]. The estimate $\hat{x}(t)$ can be used, in place of $x(t)$, to solve the optimization problem (7.5).

The adopted KF has a twofold functionality. First, as we just described, based on the dynamic model (7.1), it computes a state estimate $\hat{x}(t)$ that the MPC uses to compute the next control action. Second, it is adapting the state estimate to the actual behaviour of the system. This is relevant for a number of reasons: the controlled system may change its behaviour over time, there might be unpredictable disturbances acting on the system, or the system is not following the linear dynamics of the dynamic model (7.1). In all the cases, the KF is adapting online the choice of the estimate $\hat{x}(t)$, returning a value that is compatible with the input-output behaviour of the running system, as if it was described exactly by the dynamic model (7.1) [Lju99].

The block diagram for the resulting control scheme is represented Figure 7.1.

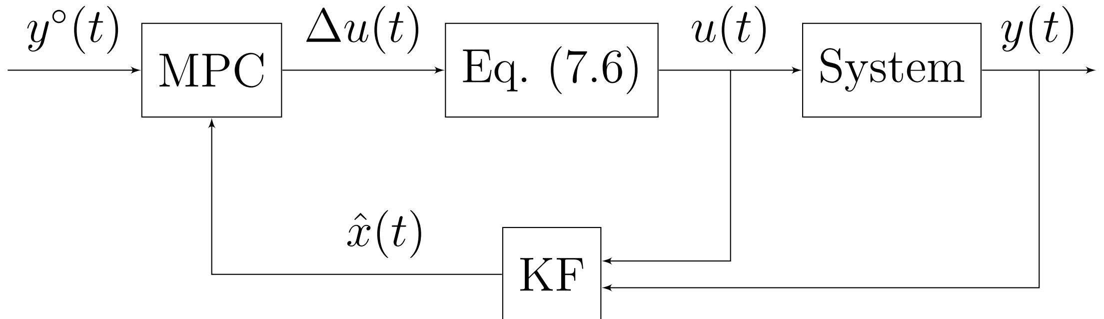


Figure 7.1: Control scheme.

7.1.2 Formal guarantees

Applying Control Theory to software systems provides a set of formal guarantees about the quality of the adaptation process [FMA+15]. The MPC adopted in this work belongs to a class of controllers named *optimal controllers*, since the computation of control decisions is based on the solution of an optimization problem. In particular, the MPC accounts for model predictions in order to make optimal adaptation plans with respect to system requirements, and compliance to the requirements about the adaptation process itself [ASM14].

The formal guarantees for the MPC have as follows. First, it is possible to ensure that all the goals are reached, subject to actuators constraints, i.e. there exists a value of the actuators within the given constraints specified in the optimization problem (7.5) that allow the system to reach its goals. If this is not the case, due to the optimal nature of the controller, the MPC finds a configuration for the actuators that minimizes the distance between the indicators and the goals. Such a distance depends on the chosen weights for each indicator in the cost function of the optimization problem (7.5).

Furthermore, since the cost function accounts for a time horizon, it is possible to guarantee that the convergence time is minimum. The dynamic

model (7.1) relates control parameters and indicators including the dimension of time. Therefore, the adaptation mechanism is able to drive the system to the goals as soon as possible, as specified by the cost function of the optimization problem (7.5). Moreover, the optimization problem (7.5) can be easily extended in order to account for additional constraints, such as for example ones on the indicators. *AwReqs* and *AdReqs* impose such constraints over the elicited goals and the adaptation process respectively which must be taken into consideration when a new adaptation plan is produced.

The MPC formulation is well suited for addressing also real-time issues and have been applied to various domains, such as aircraft control [QB03, HJS+14a]. Since the proposed solution requires a solution to an optimization problem at each control instant, it is critical to discuss possible such issues. In many cases, in fact, the time required for computing the value of the next control action might be longer than the time between two subsequent control actions. In order to overcome this challenge, there is significant literature in the control community on how to implement fast solvers [JKC12, Gis14], especially for embedded systems [JGR+14], possibly co-designing also a dedicated hardware for the solution in case of critical systems [HJS+14b]. An overview on the matter can be found in [ZRD+14].

In many cases such kind of advanced algorithms are not required when dealing with software components, and for the most critical applications some modification to the control problem can help in reducing the complexity. For example, one way to reduce the complexity is to set $\Delta u_{t+1} = \Delta u_{t+2} = \ldots = \Delta u_{t+H-1} = 0$, and let the optimization problem decide only the value for Δu_t, i.e. the one that will be actually applied to the system. This modification reduces the complexity to $\mathcal{O}(m^3)$.

Another way to deal with real-time issues is to exploit simple properties

of interior point algorithms. In fact, the solution is obtained in a fixed amount of steps with an iterative method. The current solution is always a suboptimal yet feasible solution to the optimization problem. This means that if the iterative method did not converge before a new control action is required, it can be forced to stop and return the current sub-optimal solution. This allows the controller to fulfil real-time deadlines.

Finally, another possibility to deal with real-time deadlines is exploiting the proactive nature of the MPC. As we mentioned earlier, the MPC is computing at each iteration step a plan of future actions Δu_{t+i-1}, $i = 1, \ldots, H$, then according to the receding horizon principle, only the first one is applied, i.e., $\Delta u(t) = \Delta u^{\star}_t$. Assuming that at the next control instant, the solver takes more time to converge and that a new control action is required before the optimal solution is found, one can store the previously computed plan and apply the second control action, i.e., $\Delta u(t+1) = \Delta u^{\star}_{t+1}$. This is obviously suboptimal, since it neglects the last information about the measured output, but it is able to fulfil the real-time deadlines.

7.2 Design phase

Our approach starts with the elicitation of all kinds of requirements about the target system. When all goals are refined, *AwReqs* are assigned to those that are considered most important and prone to failure. An *AwReq* AR_i defines a reference goal $y^{\circ}_i(\cdot)$ for the controller's output. Table 7.1 enlists all the reference goals for the Meeting-Scheduler exemplar as depicted in Figure 7.2.

As we mentioned earlier, the constraints imposed by *AwReqs* are not always feasible or might become infeasible in the future. For instance, the prices of hotel rooms rise every year and consequently $I1$ will fail more often as time passes. Alternatively, during summer prices are usually higher.

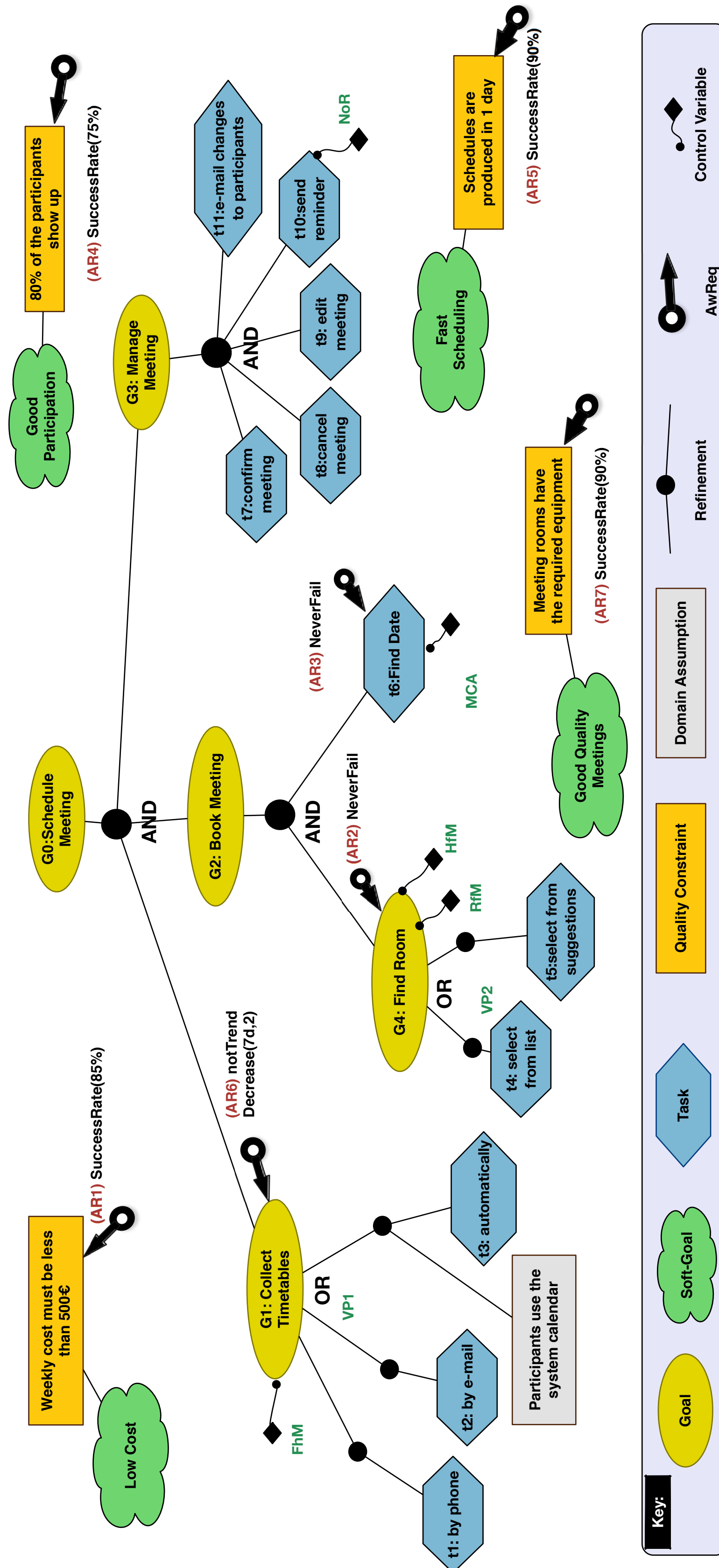


Figure 7.2: Meeting Scheduler goal model

Table 7.1: Reference goals

AwReq	$y^{\circ}(\cdot)$
$AR1$	$y_1^{\circ}(\cdot) = 85$
$AR2$	$y_2^{\circ}(\cdot) = 100$
$AR3$	$y_3^{\circ}(\cdot) = 100$
$AR4$	$y_4^{\circ}(\cdot) = 75$
$AR5$	$y_5^{\circ}(\cdot) = 2$
$AR6$	$y_6^{\circ}(\cdot) = 90$
$AR7$	$y_7^{\circ}(\cdot) = 90$

Hence, stakeholders could accept a lower success for $I1$ (in other words $y_1^{\circ}(\cdot) < 85$). At this step of the design phase, the domain experts, along with the stakeholders, analyze and evaluate such conditions and specify *EvoReqs* for the system-to-be. The *EvoReqs* operations defined for the *AwReqs* of the Meeting-Scheduler are presented in Table 7.2.

Table 7.2: *EvoReqs* operations

AwReq	*EvoReq* operation
$AR1$	1. Relax($AR1$,$AR1'$_75) 2. Strengthen($AR1$,$AR1'$_85)
$AR2$	Relax($AR2$,$AR2'$_90)
$AR3$	Relax($AR3$,$AR3'$_90)
$AR4$	1. wait(3 days) 2. Relax($AR4$,$AR4'$_75)
$AR5$	Replace($AR5$,$AR5'$_3)
$AR6$	wait(3 days)
$AR7$	wait(2 days)

When summer season begins and hotel prices are higher, the first *EvoReq* operation is triggered relaxing the reference goal from 85% to 75%. The second *EvoReq* operation is triggered when summer season ends and the threshold is restored to its previous value. Similarly, when $AR2$ and $AR3$ fail for more than 2 days in a row, the reference goals are relaxed for a

week[1]. In case of $AR5$, when goal $G1$ tends to fail more than 2 times/week, the constraint is permanently replaced by 3 times/week. Finally, when $AR6$ and $AR7$ fail for more than 2 days the adaptation mechanism ignores them for 3 and 2 days respectively.

Next, by applying AHP, weights are elicited for each indicator to capture their relative importance. As a rule of thumb, indicators assigned to functional requirements have higher priority compare to non-functional ones. These weights are the values of matrix Q of the cost function. The controller, through the optimization function, finds an equilibrium for every goal, putting more effort on fixing the most important ones. As for the control parameters, their weights are empirically elicited assigning lower weights to the control parameters we want to be tuned less often. These weights are the values of matrix P of the cost function. In our exemplar, for instance, increasing the number of rooms RfM is preferred over HfM since it is a less costly solution, does not require any authorization and, therefore, takes effect immediately. The elicited priorities for the indicators of Meeting-Scheduler and the weights for control parameters as shown in Table 7.3 and Table 7.4, respectively.

Table 7.3: Indicator Priorities

Indicator	Priority
$I1$	0.15
$I2$	0.3
$I3$	0.3
$I4$	0.06
$I5$	0.2
$I6$	0.05
$I7$	0.04

The last set of requirements to be elicited are the *AdReqs*. These re-

[1] The relaxation duration and the triggering condition are prescribed by the stakeholders.

Table 7.4: Control Parameter weights

Control Parameter	Weight
FhM	1
MCA	1
RfM	1.2
HfM	0.6
NoR	1.2
$VP1$	0.8
$VP2$	1.4

quirements impose constraints to the adaptation process itself. For the particular case of MPC, an *AdReq* specifies the receding horizon of the controller and, consequently, how far in the future the adaptation plan should target. Other *AdReqs* might refer to the magnitude of allowed change of control parameters. For instance, HfM cannot be increase more than 5 units each time.

Finally, a quantitative model such as that in Equation 7.1 must be derived. Given the absence of laws of nature we ran a long simulation of the meeting scheduler system during which the control parameters change often and both control input and output are recorded. With the aid of Matlab and System Identification toolbox [2] we estimate the analytical model of the system. Even if the system-to-be cannot be simulated accurately, the model can be improved later on, when the real system is deployed, by means of a learning mechanism during the runtime phase, which is explained in detail in the next chapter.

7.3 Chapter Summary

In this chapter we illustrate the basic components of an MPC controller and how these components are related to *AwReqs*, indicators and control

[2] http://it.mathworks.com/products/sysid/?requestedDomain=www.mathworks.com

parameters, as we have examined them in the previous chapters. More specifically, we presented how an analytical model derived using system identification can be used to describe the dynamics of a software system, including the time dimension. Next, we explained how MPC composes adaptation plans over a predefined horizon, by solving a multi-objective optimization problem. We also demonstrated the use of Kalman filters in order to cope with unavoidable nonlinearities of the system.

Chapter 8

Control-based software adaptation

> The voyage of discovery is not in seeking new landscapes but in having new eyes.
>
> Marcel Proust

In chapter 5 we presented a qualitative adaptation mechanism in order to handle conflicting goals under the absence of analytical models that describe the system's behaviour. Next, in chapter 6 we demonstrated how the next adaptation problem can be treated as multi-objective optimization problem. In this case, we assumed the existence of quantitative relations that describe the impact of control parameters over indicators. However, this approach does not take into account the dimension of time. Moreover, the analytic model that we used in chapter 6 might suffer from inaccuracies and nonlinearities. Therefore, in chapter 7 we presented a control theoretic approach for finding optimal adaptations with the use of a MPC.

This chapter illustrates an adaptation framework that has as main component an MPC controller designed as prescribed in the previous chapter. This framework provides and answer to **RQ5:** *How to find an optimal adaptation under the absence of any information about system's dynamics?*

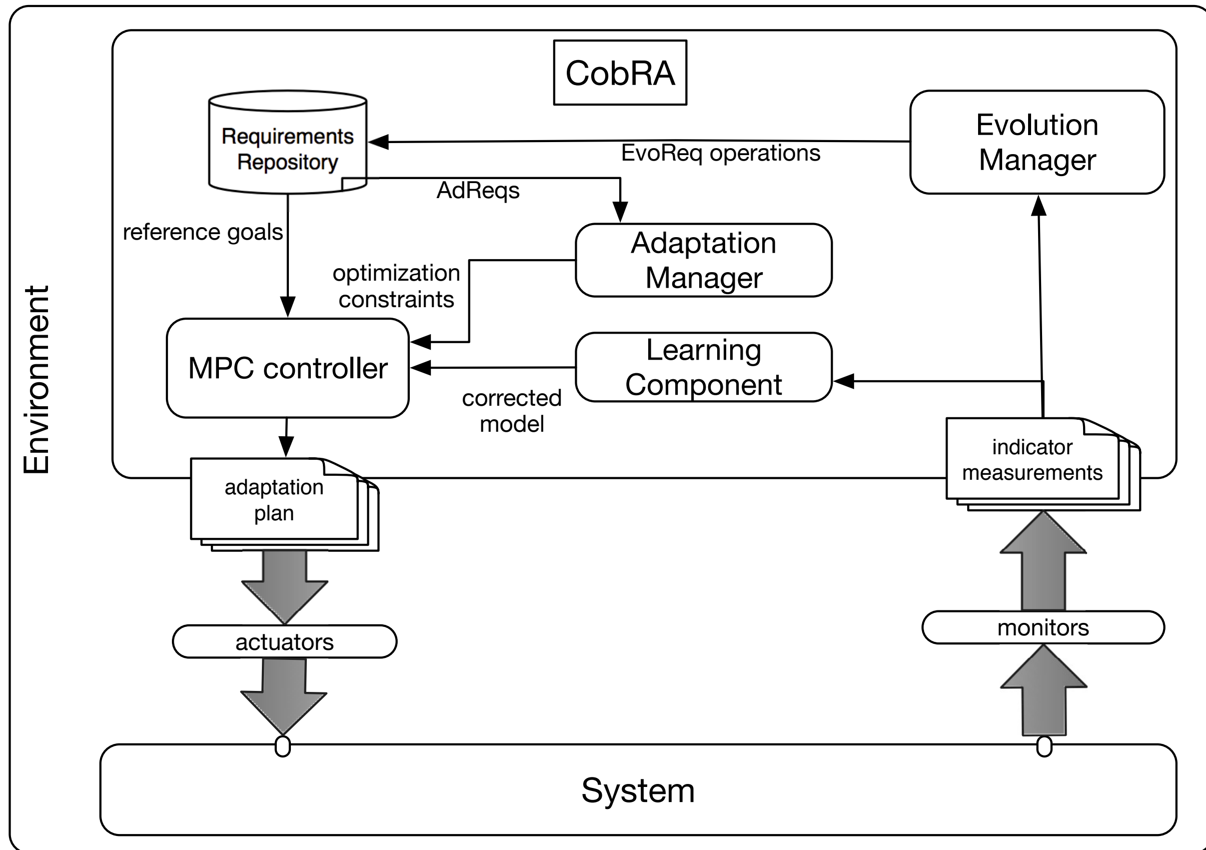


Figure 8.1: CobRA framework

8.1 The CobRA framework

When the design phase is completed and the system is implemented, the *CobRA* (Control-based Requirements-oriented Adaptation) framework can be deployed and play the role of the adaptation mechanism. *CobRA*, depicted in Figure 8.1, has five main components. The monitors and the actuators that integrate *CobRA* with the target system are application specific and must be implemented by the designers of the system.

Requirements repository. This repository stores all the models produced during the design phase and provides information to the other components of the framework when requested.

Evolution manager. This component analyzes the logs provided by the monitors in order to identify conditions that would trigger *EvoReq* operations. If a requirement is replaced either permanently or temporarily, it updates the requirements repository.

Adaptation manager. This component translates *AdReqs* to constraints for the optimization problem of Equation 7.5. Such constraints are related to maximum allowed decrease or increase of a control parameter in a single step and the weights of all indicators and control parameters (matrices Q and P).

Learning component. Black-box system identification does not always provide precise models about the system's behaviour. Therefore, we include in our framework a learning component that, based on the applied changes and the outcome values of indicator that occurred as a result of these changes, revises the control law to adapt to changes of the behaviour of the system. More specifically, this component is an implementation of the Kalman Filter as it is described in the previous chapter.

MPC controller. The details of this component have been discussed in the previous chapter. Summarizing its functionalities, the MPC controller requests the requirements repository for the reference goal $y^{\circ}(\cdot)$ of each indicator monitored. It then calculates the distance of each indicator from its respective reference goal and composes an adaptation plan that minimize every distance taking into account the indicator priorities in order to restore equilibrium, subject to the given constraints on the control parameters. The plan includes changes to control parameters in a predefined horizon. For example, the indicators of the Meeting-Scheduler are evaluated daily and the plan includes values for control parameters so that indicators minimize their distance from $y^{\circ}(\cdot)$ for the next three days. If two days after the plan is applied the result is not what was expected, e.g., because the number of meetings constantly grows, the controller produces

a new plan which tries to anticipated future failures in a receding horizon fashion.

The iterative adaptation process with *CobRA* includes the following steps:

1. **Step 1:** The monitors collect the measurements of all the indicators of the system.

2. **Step 2:** The *Evolution Manager* examines if any event that would trigger an *EvoReq* operation is present and in that case updates the evolved requirement in the Requirements Repository.

3. **Step 3:** The Adaptation Manager provides the MPC controller with the weights for the indicators and control parameters as well as constraints for the optimization problem.

4. **Step 4:** The Learning Component provides the MPC with a corrected model of the system based on the recent measurements.

5. **Step 5:** The MPC controller given the current reference goals provided by the Requirements Repository, and the corrected model produces a revised adaptation plan with the target each indicator value to converge to the reference goal within the prediction horizon.

6. **Step 6:** The actuators apply the first step of the plan to the system.

It is important to mention that if an a new requirement is introduced or an older one is removed the design phase must be repeated in order to derive a new analytical model.

8.2 Evaluation

In the previous sections we provided the basic background for the structure and functionalities of an MPC controller. We also presented the *CobRA*

framework that exploits stakeholder goals and uses an MPC controller to compose dynamically adaptation plans when requirements are not met. In this section we evaluate and compare *CobRA* with *Zanshin* that also has as its baseline for adaptation stakeholder requirements and adopts concepts from Control Theory.

8.2.1 Methodology

We have conducted our experiments with a simulation of the Meeting-Scheduler exemplar[1] implemented in Python and ran on a computer with an Intel i5 processor at 2.5GHz and 16GB of RAM. We ran the simulation for 10.000 steps while automatically modifying all the control parameters which must cover all the range of their potential values. The result of this process is a log file with all the values of the inputs and outputs of the system at every step. Then, we executed once a Matlab procedure from the Matlab System Identification Toolbox in order to estimate an analytical model that describe the system's behaviour as it is described in Section 2.2.

After acquiring the system's quantitative model, we stress-tested the simulation by modifying various environmental parameters such as the user's availability, punctuality and the number of meeting requests that must be scheduled every day. At this phase, we tune the controller by modifying the weights of the outputs and the inputs. If an indicator, especially a not very important one, constantly overshoots, its weight must reduced. Similarly, if a control parameter that we do not wish to change often, such as the number of hotel rooms available, the associated weight must be increased. As a rule of thumb the user must keep the weight values in the same magnitude. Moreover, the order of the modified weights

[1] `https://gitlab.com/konangelop/it.unitn.disi.konangelop.simulations.meeting_scheduler_v2.git`

of the indicators must be compliant to the one the stakeholders provided. When the MPC controller reaches a desired behaviour, the tuning phase is completed.

Zanshin as opposed to *CobRA* does not involve any quantitative models, but only qualitative relations between inputs and outputs based on human expertise. For example, it is known by the domain expert that by increasing MCA the value of $I3$ increases. For our experimentation we used the default adaptation algorithm of *Zanshin* as described in [SLM12b]. When a failure arrives *Zanshin* randomly selects a control parameter that will improve this failure and increases or decreases it by a predefined amount. Therefore, we provided such qualitative information to *Zanshin* based on a previous studies of Meeting-Scheduler.

For the evaluation and the comparison of the two frameworks, we put the simulated system under a stress-test and we compare the behaviour of the outputs in each case. We also compare the values of the the cost-function described in Section 7.1 through time for both frameworks, comparing which minimizes it most. The selection of *Zanshin* for the purposes of our evaluation is based on two reasons. First, it uses the same requirements-based monitoring mechanism using *AwReqs* as *CobRA* and therefore, customization of the adaptation problem was not required. Second, *Zanshin* decides adaptation plans based on qualitative information provided by domain experts, while *CobRA* uses an automatically derived quantitative model that captures the dynamics of the system.

The Meeting-Scheduler application receives daily a number of meeting requests. Once the timetables are collected, a date for each meeting must be found. The result of the finding date process is pseudorandom, given that it depends on control and environmental parameters that change based on stochastic processes we have encoded in the simulation. For instance, as the availability of the participants drops, the more often the goal *Find Date*

will fail. Similar pseudorandomness has been encoded for other goals such as *Find Room* and *High Participation*. For the purpose of our experiment we run the simulation for 60 steps (simulation days), during which the number of meeting requests gradually increases and then decreases along with the participants availability. Hereby, we present the results only for indicators $I1$-$I4$.

8.2.2 Experimental Results

Figure 8.2 depicts the values of the indicators at each step of the simulation. As the number of meetings grows the cost for the system increases as well, resulting in the decrease of the indicator $I1$. However, $CobRA$ though manages to recover by preferring local rooms over hotel rooms as it can be seen in Figure 8.3, whereas $Zanshin$ fails to restore the failure. As it concerns indicator $I2$, $CobRA$ converges almost immediately, whereas $Zanshin$ requires considerably more time. The reason of the delay is that $Zanshin$ increases its control parameters by a fixed amount rather than basing it on the magnitude of failure as $CobRA$ does. In the case of $I3$ the human expertise provided to $Zanshin$ matched the identified relation we derived experimentally for $CobRA$, since the two frameworks achieved almost identical values. Finally, for indicator $I4$ $CobRA$ outperforms $Zanshin$, by increasing NoR more than the latter.

The last metric we use for our evaluation is the cost-function $\hat{J}(t) = (y^{\circ}(t) - y(t))^T Q(y^{\circ}(t) - y(t)) + \Delta u(t)^T P \Delta u(t)$. The value of the cost function is calculated from the measured values of the indicators and the changes performed over the control parameters. In Figure 8.4 we present the individual value of the cost function at each step the simulation, on the top and the cumulative cost $\sum_t \hat{J}(t)$ on the bottom. $\hat{J}(t)$ captures the magnitude business value loss because of failing indicators and adaptation costs for changing control parameters. $CobRA$ minimizes more at each

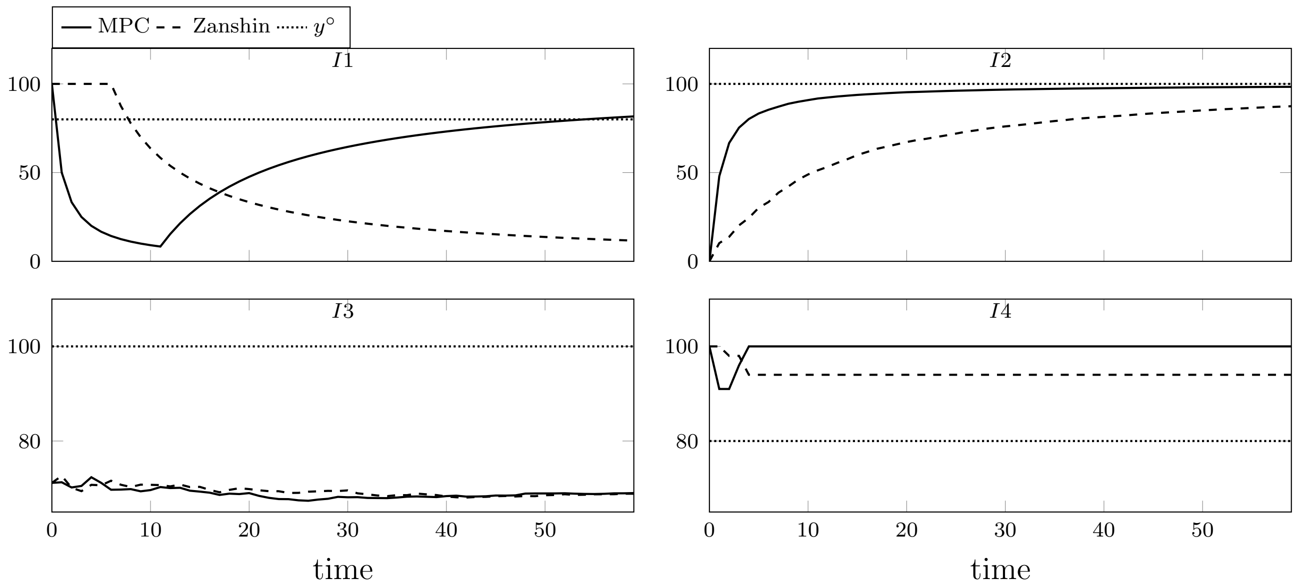


Figure 8.2: Indicator measured values

step the cost function and by the end of the simulation it produces more stable results. On the other hand, *Zanshin*'s adaptation results in higher losses at most steps, while the accumulated value of the cost-function is growing monotonically. The minimization of the cost over the simulation is highlighted in the cumulative cost showed in the bottom graph of Figure 8.4.

8.2.3 Discussion

From the experimental results we can safely assume that *CobRA* can produce adaptation plans that allow the system to recover faster from failures while maintaining an equilibrium among conflicting requirements. Moreover, our framework outperformed the qualitative adaptation of *Zanshin* in most cases and proved that Control Theory can be applied to generic software systems such as the Meeting-Scheduler exemplar.

Another contribution of *CobRA* is that it managed to adapt even if the underlying system has nonlinear behaviours. The used simulator, in fact,

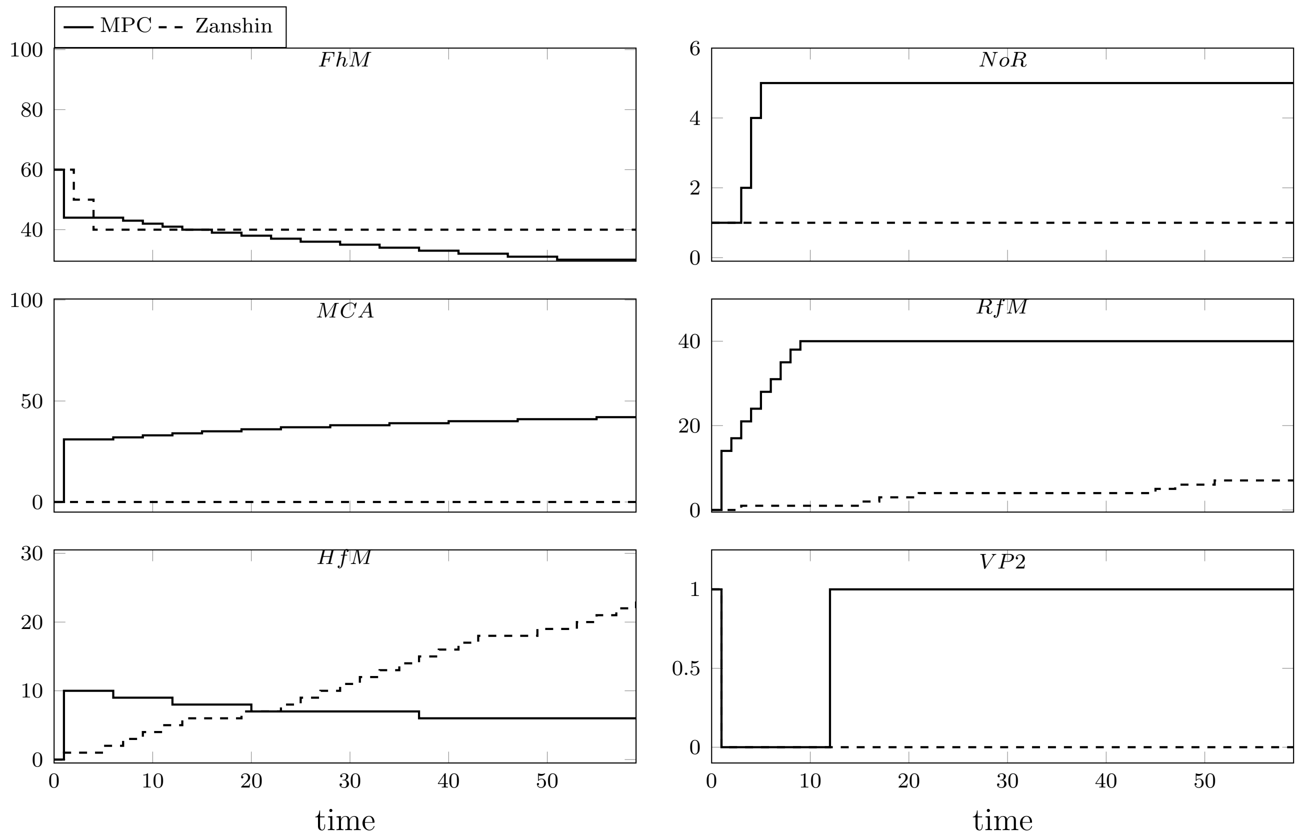


Figure 8.3: Control parameter values

includes also nonlinear relationships between inputs and outputs, to obtain more realistic behaviours. In practice most systems have input-output relations that are nonlinear and therefore it is important for an adaptation mechanism to handle efficiently model imperfections. In particular, the KF contributes to correcting the model as the system runs, allowing MPC to make more accurate predictions.

For the Meeting-Scheduler exemplar a linear model was sufficient for predicting the system's behaviour. However, this might not be always the case. For systems with nonlinear dynamics, either tailored models can be used [PMTL15], or more advanced system identification techniques are available [Lju10, Lju99], and nonlinear MPC formulations can be adopted

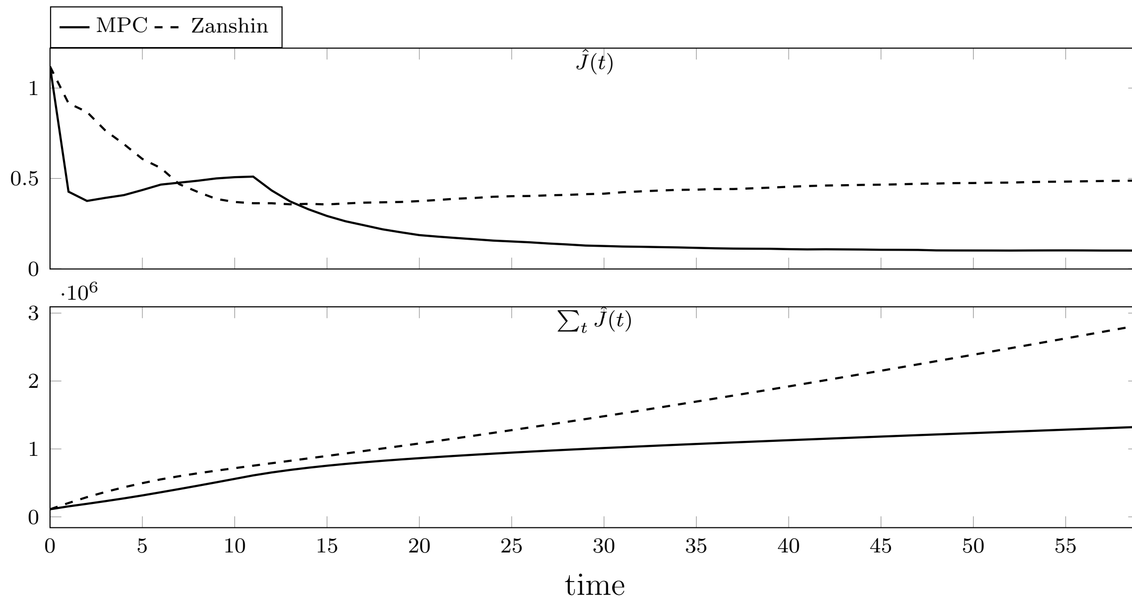


Figure 8.4: Adaptation cost

[AZ00]. As future work we intend to evaluate further our approach using more complex systems, identify their particularities and apply variations of MPC to deal with them.

A main drawback of *CobRA* is that it requires a simulation or historical data of the system in order to derive the analytical model it needs to operate. This is not always possible since for software systems there are no methodologies yet, as for physical systems, to guide the system designers simulate a model that can produce data sufficiently similar to those of the real system.Developing such methodologies for software engineers, as well as establishing guidelines for tuning the MPC parameters, as described in chapter **??**, open new research agendas that go beyond the scope of this thesis. However, relating concepts such as *AwReqs*,*EvoReqs* and *AdReqs* with the basic elements of control engineering is a first step towards this direction.

Finally, it is worth noting that some control parameters of our exemplar, such the number of rooms, are discrete, but according to the equation 2.1

control parameters are continuous variables. In Control Theory this problem is known as the actuation design problem. There are two different approaches a) using rounding of the continuous variable computed by the MPC and b) by adopting a Pulse Width Modulation-like policy [MHP+12].

8.3 Chapter Summary

The main contribution of this chapter is adopt the concept of an MPC to the design of self-adaptive software systems. To accomplish this, we propose a framework, named *CobRA*, that integrates MPC components with previous work on software engineering for self-adaptive systems. We also provide guidelines on how to tune the variables of the MPC controller for better results during the adaptation process.

The distinct feature of *CobRA* compared to other approaches is the use of an analytical model to capture the relationship between the control parameters and the output of the system. This model can accurately predict the system's behaviour and allows *CobRA* to react to environmental changes and compose dynamically adaptation plans. The analytical model is the product of an automated system identification process, capturing relations that human experts might not be aware of. We evaluated our framework using an implementation of the Meeting-Scheduler exemplar and compared the result to those of *Zanshin* framework. The results of our evaluation show that control-theoretic concepts can be very effective in producing adaptation plans for software systems and most of the times provides better results than human experience-based approaches.

Chapter 9

Conclusions and future work

> The only way to be truly satisfied is to do what you believe is great work, and the only way to do great work is to love what you do. If you haven't found it yet, keep looking, and don't settle. As with all matters of the heart, you'll know when you find it. And like any great relationship, it just gets better and better as the years roll on. So keep looking, don't settle.
>
> Steve Jobs

In this thesis we proposed a novel approach for designing self-adaptive systems that is founded on high variability models and a set of frameworks that are apply control-theoretic techniques. Throughout its chapters we presented a methodology for eliciting large adaptations spaces that allow software systems to tackle environmental uncertainty. Next, we proposed a qualitative adaptation mechanism and the use of *Adaptation Requirements* in order to compensate for the absence of analytical models for handling multiple failures. This part of our work includes an extension of the *Zanshin* framework and is useful when no quantitative model can be elicited, neither empirically nor by simulation. Then, we propose two more adaptation mechanisms, one based Optimization Modulo Theories and Constrained Goal Models, named *Prometheus*, whereas the other,

named *CobRA*, is based on Control Theory and in particular Model Predictive Control. *Prometheus* is suitable in cases where empirical quantitative models can be provided in advance by domain experts or after using the extended Zanshin for a a certain amount of time. Moreover, *Prometheus* is suitable in cases where there are constraints among goals. On the other hand, when a simulation of the system-to-be can be provided, *CobRA* can provided adaptations of enhanced precision. The reason is that empirical models usually suffer from inaccuracies, which are not handled at runtime by *Prometheus* as opposed to *CobRA*. Each of our approaches is evaluated with a well known exemplar in Software Engineering literature, namely the Meeting-Scheduler.

In this chapter we conclude this thesis summarizing the contributions of our work as well as its limitations. Finally, we present a list of ideas that constitute our future work.

9.1 Contributions to the state-of-the-art

The contributions of this thesis provide answers to the research questions we presented in Chapter 1 and we list below:

RQ1: How does an adaptation space based on requirements relates to architecture-based adaptation spaces?

RQ2: Can we extend existing techniques to relate requirement-based adaptation spaces to other aspects?

RQ3: How do we deal with multiple failing requirements under the absence of quantitative information that describe the system dynamics?

RQ4: How could the self-adaptation problem be formulated as an optimization problem and how could it be solved?

RQ5: How to find an optimal adaptation under the absence of any information about system's dynamics?

The presence of multiple approaches in the literature of self-adaptive systems motivated us to investigate their common characteristics, but most importantly their differences. The main distinction point we identified was the kind of models each approach uses to capture its adaptation space. The two most applied kinds of models where requirement and architecture models. Hence, we decided to select two representative frameworks, one that its adaptation space is based on requirements and another on architecture, *Zanshin* and *Rainbow* respectively.

For the comparison of the two frameworks we used a load balancing exemplar where the main objectives of the system are to serve web content while maintaining a certain level QoS. Even though the experimentation has shown prominent results for both frameworks, the focus of our comparison was on conceptual level rather than technical. More specifically, the comparison outcome has shown that requirement based approaches document better the objectives of the system, but might overlook architectural constraints and solutions that are not available before the system's design. On the other hand architecture based approaches encode system requirements in their adaptation strategies without taking into account that these might change under certain circumstances, as in the case of Evolution Requirements. However, architecture adaptation spaces consider dependencies among the architecture components and variability at the lower level of the system. Therefore, the overall conclusion of this com-

parison was that combined adaptation spaces can improve the adaptation process. This result has later been verified by Chen et al. in [CPY+14].

Guided by the outcome of our comparison, we developed a Three-Peaks process that combines architecture and requirement adaptation spaces. In addition to these dimensions we include that of behaviour. The purpose of our process is to guide the designers of self-adaptive systems to design high variability adaptation spaces that are able to tackle the environment's uncertainty. The process starts by eliciting system goals and in an intertwined manned make decisions about the components that will carry them out as well as the system's behaviour. As requirements, behaviour and architecture are refined we identify *AwReqs* that need to be monitored and parameters of the environment that could cause failures. Therefore, the aim of our process is to elicit sufficient number of control parameters in each of three dimensions of the adaptation space allowing always the system to recover when failures take place, combining the advantages of the existing requirements, architecture and behaviour-based adaptation approaches in the state of the art.

Our work on designing high variability self-adaptive system improves the state-of-the-art by combining the three complementary dimensions of software systems, requirements, architecture and behaviour. Comparing our work to STREAM-A [PLC+12], the latter omits the essential dimension of behaviour and the heurestics provided for eliciting architectures from goal models aim to reflect the system's requirements on components, ignoring the impact of architectural decisions on the requirements. The work by Sykes et al. [SHMK08], takes into account the intertwined relationship of the three dimensions in a hierarchical manner. When the architecture fails to fulfil its mandate, an architectural adaptation is applied. If this is not possible, a new behaviour for the system is decided. In case a new behaviour is not feasible, an adaptation at requirements level

takes place. However, this approach as opposed to ours provides only a reference framework that exploits existing variability to produce adaptations and not producing sufficiently large adaptation space to cope with uncertainty.

Next, we proposed a qualitative adaptation mechanism that extends *Zanshin* and uses qualitative system identification to capture the impact of control parameters on *AwReqs*. In this piece of work we investigated the various types of conflicts that occur among system goals and introduced a new kind of requirements, namely Adaptation Requirements in order to define policies that allow efficient trade-offs. This work extended the *Qualia* process [SS12], which considered only the case of SISO systems, whereas our extension makes it possible to handle MIMO systems as well.

In Chapter 6 we defined the Next Adaptation Problem as a constrained multi-objective optimization problem. Moreover, we illustrated that an adaptation space is not static and it depends on the availability of the implemented alternatives. In other words, certain alternatives can be applied only if certain assumptions hold. Hence, we proposed a framework that is built on the top of a diagnostic tool that detects failures and available solutions composing a new adaptation each time one or more goals are not met. The proposed framework minimizes the degree of failure for every hard goal while optimizing quality attributes related to soft goals.

The extended *Zanshin* and *Prometheus* compared to other approaches [Che08, SHMK10, SMK11, ZSL14] presented in Section 2.3 that use priorities (referred also as utilities by the related work), our frameworks deal with failures of both functional and non-functional requirements, whereas the other approaches focus only on non-functional requirements. Another distinct difference is that *Prometheus* provides a guide to the designers in order to define the adaptation problem as a Multi-Objective Optimization problem. Moreover, *Zanshin* and *Prometheus* compose adaptation strate-

gies dynamically, whereas *Rainbow* [Che08] only provides predefined adaptation strategies, automating human administration processes. Finally, along with the implementation of *Prometheus* we propose guidelines on how the designers can elicit the cost-functions that the adaptation framework needs to optimize.

The last part of our approach integrates software engineering for self-adaptive systems with Control Theory. More specifically, we introduced the use of Model Predictive Control, a control-theoretic technique, in order to develop an adaptation mechanism that minimizes the control error of each monitored goal by selecting the less costly adaptation. This framework requires the use of analytical models that are derived through formal system identification. Such model are characterized by inevitable inaccuracies and nonlinearities and therefore we included a Kalman filter in order to tackle these obstacles.

Comparing our work to later versions of *Rainbow* [CGSP15] where the authors propose the use of Probabilistic Model Checking in order to compose strategies dynamically, Our approach differs to theirs as it requires precise knowledge of how each control parameter of the system influences the output of the system, such as, adding one server improve response time by one second. In systems with dynamic environments such relations might change over time and are not always linear. CobRA's MPC uses the derived analytical model to reason about the impact of changing a control parameter instead of human experience and overcomes nonlinearities by applying a Kalman filter.

Compared to [FHM14] which provides a control-based approach which treats SISO systems by varying a single input and measuring the output, *CobRA* is more advanced since it can deal with MIMO systems. Next, another approach that are proposed for controlling MIMO systems [FHM15], where the MIMO control is obtained as an automated synthesis by compos-

ing SISO controllers in a hierarchical way. The approach has the limitation that the influence of different control parameters on the indicators is not included in the model and it is treated only coupling a single control parameter to a specific indicator. On the other hand, *CobRA* includes all the mutual influences in the single model used for the control design. This can be exploited when deciding the values of the control parameters in order to obtain a better adaptation plan. Moreover, in our work, as opposed to the control-based approaches presented in Section 2.3, we integrate control design and requirements engineering in order to provide a guideline about to how to integrate MPC with the development of self-adaptive software.

9.2 Limitations of the approach

While our approach provides high variability models and advanced adaptation techniques that can efficiently deal with conflicting requirements and multiple failures, it is limited in several aspects.

- **Centralized control.** For every adaptation mechanism we proposed, we assumed that is able to control the entire system. However, very large systems, where their components are highly distributed require separate control mechanisms that are able to communicate. Another limitation of centralized control is that the components loosely coupled systems such as service oriented systems, are not aware of each other's goals and architectural constraints.

- **Case tool.** The Three-Peaks approach requires tool support in order to handle large models. We have started building an Eclipse-based tool that allows the designer to keep track of all the *AwReqs* and their conflicts, providing suggestions for additional refinements.

- **Framework prototype.** Both *Prometheus* and *CobRA* have been developed with the purpose of evaluating the proposed adaptation

mechanisms. However, all the required models graphical and analytical must be created manually by the user. Therefore, components that facilitate the model derivation process must be included.

- **Controller Design Overhead.** The *CobRA* framework requires an analytical model in order to produce adaptation plans. However, this model is only derived if a simulation of the system is available. Therefore, the system identification introduces an overhead in the design of self-adaptive systems.

- **Experiments.** Every part of our approach has been evaluated with a simulation of the Meeting-Scheduler exemplar [1]. This system is a good fit for illustrating all the aspects of our research proposal. Moreover, it is a generic kind of software system, that involves goals such as performance, cost etc. that are common in most kind of systems. However, every domain raises individual challenges and therefore further experimentation is required for the evaluation of our approach with larger case studies from various domains.

9.3 Future work

The limitations illustrated in the previous section extend our research agenda in various ways. First, we are working on the implementation of a graphic tool that supports the Three-Peaks process. This tool as we mentioned in the previous section will provide indications to the designers about potential conflicts and failures that might occur at runtime. After the completion of the tools development we plan to evaluate the effectiveness of the Three-Peaks process by a controlled experiment with master students that will be divided in two groups. Both groups will be given the

[1] https://gitlab.com/konangelop/it.unitn.disi.konangelop.simulations.meeting_scheduler_v2.git

same case study description, one will model requirements, architecture and behaviour following a sequential and not systematic process while the other one will be guided to use the Three-Peaks Case tool. The purpose of this experiment is identify which of the two groups produced larger adaptation spaces and resolved better requirement conflicts.

Another aspect we are interested in investigating is applying various control architectures such as decentralized, distributed and hierarchical [Sca09] to deal with large scale systems. These architectures include local controllers for individual components that are able to communicate and coordinate for achieving common objectives.

Our future work includes further experimentation and evaluation of our approaches with case studies produced within the research community of self-adaptive systems [2]. Moreover, we plan to compare our proposed frameworks with those that have already been applied for these case studies to identify new opportunities for improvement.

As we mentioned in Chapters 7 and 8, Control Theory is a discipline with which only a few software engineers are familiar. Therefore, the software lifecycle model should be revisited to include tasks such as simulation, system identification and controller design. These new tasks must be supported by tools that will allow software engineers to design self-adaptive applications with the use of control techniques without the need of understanding the heavy formalisms behind them.

Finally, the literature in the self-adaptive systems field offers multiple approaches for producing adaptation strategies. Hence, we are interested in performing a literature review where we can identify and categorize the software systems with respect to certain characteristics, e.g. application domain, continuous or discrete time, real time systems etc. The purpose of such a review is to identify which decision-making mechanism is most

[2] https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/

suitable for each category based on its characteristics.

Bibliography

[ABG+13] Aldeida Aleti, Barbora Buhnova, Lars Grunske, Anne Koziolek, and Indika Meedeniya. Software architecture optimization methods: A systematic literature review. *Software Engineering, IEEE Transactions on*, 39(5):658–683, 2013.

[ADH+08] Tarek Abdelzaher, Yixin Diao, Joseph L Hellerstein, Chenyang Lu, and Xiaoyun Zhu. Introduction to control theory and its application to computing systems. *Performance Modeling and Engineering*, pages 185–215, 2008.

[AG94] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 71–80, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[ASM14] Konstantinos Angelopoulos, Vítor E. Silva Souza, and John Mylopoulos. Dealing with multiple failures in zanshin: a control-theoretic approach. In *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Proceedings*, SEAMS 2014, pages 165–174, 2014.

[Ast95] Karl J Astrom. Pid controllers: theory, design and tuning. *Instrument society of America*, 1995.

[AZ00] Frank Allgöwer and Alex Zheng. *Nonlinear model predictive control*, volume 26. Birkhäuser Basel, 2000.

[BMSG+09] Yuriy Brun, Giovanna Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi M üller, Mauro Pezzè, and Mary Shaw. Software engineering for self-adaptive systems. chapter Engineering Self-Adaptive Systems Through Feedback Loops, pages 48–70. Springer-Verlag, Berlin, Heidelberg, 2009.

[BPG+04] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[BPS10] Luciano Baresi, Liliana Pasquale, and Paola Spoletini. Fuzzy Goals for Requirements-driven Adaptation. In *Proc. of the 18th IEEE International Requirements Engineering Conference*, pages 125–134. IEEE, 2010.

[Bry09] Volha Bryl. Supporting the design of socio-technical systems by exploring and evaluating design alternatives. *University of Trento. Trento, Italy. Doctoral Thesis*, page 134, 2009.

[BSG+09] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. In *Software engineering for self-adaptive systems*, pages 48–70. Springer, 2009.

[CBA04] Eduardo F. Camacho and Carlos Bordons Alba. *Model Predictive Control*. Springer London, 2004.

[CG12] Shang-Wen Cheng and David Garlan. Stitch: A language for architecture-based self-adaptation. *J. Syst. Softw.*, 85(12):2860–2875, December 2012.

[CGS05] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Self-star properties in complex information systems. chapter Making Self-adaptation an Engineering Reality, pages 158–173. Springer-Verlag, Berlin, Heidelberg, 2005.

[CGS06a] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *Proceedings of the 2006 International Workshop on Self-adaptation and Self-managing Systems*, SEAMS '06, pages 2–8, New York, NY, USA, 2006. ACM.

[CGS06b] Shang-Wen Cheng, David Garlan, and Bradley R. Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems, SEAMS 2006*, pages 2–8, 2006.

[CGS09] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Evaluating the effectiveness of the rainbow self-adaptive system. In *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS'09. ICSE Workshop on*, pages 132–141. IEEE, 2009.

[CGSP15] Javier Cámara, David Garlan, Bradley R. Schmerl, and Ashutosh Pandey. Optimal planning for architecture-based

self-adaptation via model checking of stochastic games. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 428–435, 2015.

[Che08] Shang-Wen Cheng. *Rainbow: cost-effective software architecture-based self-adaptation*. ProQuest, 2008.

[CNW+12] Hafedh Chourabi, Taewoo Nam, Shawn Walker, J Ramon Gil-Garcia, Sehl Mellouli, Karine Nahon, Theresa A Pardo, and Hans Jochen Scholl. Understanding smart cities: An integrative framework. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 2289–2297. IEEE, 2012.

[CPY+14] Bihuan Chen, Xin Peng, Yijun Yu, Bashar Nuseibeh, and Wenyun Zhao. Self-adaptation through incremental generative model transformations at runtime. In *Proceedings of the 36th International Conference on Software Engineering*, pages 676–687. ACM, 2014.

[CSBW09] Betty H. C. Cheng, Pete Sawyer, Nelly Bencomo, and Jon Whittle. A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 468–483. Springer, 2009.

[DBHM13] Fabiano Dalpiaz, Alex Borgida, Jennifer Horkoff, and John Mylopoulos. Runtime goal models: Keynote. In *Research Challenges in Information Science (RCIS), 2013 IEEE Seventh International Conference on*, pages 1–11, May 2013.

[DGM12] Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos. Adaptive socio-technical systems: a requirements-based approach. *Requirements Engineering*, pages 1–24, 2012.

[DvLF93] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Sci. Comput. Program.*, 20(1-2):3–50, April 1993.

[EG01] W. Keith Edwards and Rebecca E. Grinter. At home with ubiquitous computing: Seven challenges. In *Proceedings of the 3rd International Conference on Ubiquitous Computing*, UbiComp '01, pages 256–272, London, UK, UK, 2001. Springer-Verlag.

[EM13] Naeem Esfahani and Sam Malek. Uncertainty in self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II*, pages 214–238. Springer, 2013.

[FDC14] Erik M Fredericks, Byron DeVries, and Betty HC Cheng. Autorelax: Automatically relaxing a goal model to address uncertainty. *Empirical Software Engineering*, 19(5):1466–1501, 2014.

[FF95] Stephen Fickas and Martin S. Feather. Requirements monitoring in dynamic environments. In *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on*, pages 140–147, Mar 1995.

[FHM14] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *36th International Conference on Software Engineering, ICSE '14*, pages 299–310, 2014.

[FHM15] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated multi-objective control for self-adaptive software design. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 13–24, New York, NY, USA, 2015. ACM.

[Fis70] Peter C Fishburn. Utility theory for decision making. Technical report, DTIC Document, 1970.

[FMA+15] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolás D'Ippolito, Ilias Gerostathopoulos, Andreas B. Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro Vittorio Papadopoulos, Suprio Ray, Amir Molzam Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel. Software engineering meets control theory. pages 71–82, May 2015.

[Gar14] David Garlan. Software architecture: A travelogue. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 29–39, New York, NY, USA, 2014. ACM.

[GC15] M. Gaggero and L. Caviglione. Predictive control for energy-aware consolidation in cloud datacenters. *Control Systems Technology, IEEE Transactions on*, pages 1–14, 2015.

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, October 2004.

[GCS03] David Garlan, Shang-Wen Cheng, and Bradley Schmerl. Architecting dependable systems. chapter Increasing System Dependability Through Architecture-based Self-repair, pages 61–89. Springer-Verlag, Berlin, Heidelberg, 2003.

[Gis14] Pontus Giselsson. Improved fast dual gradient methods for embedded model predictive control. In *IFAC World Congress*, volume 19, pages 2303–2309, 2014.

[GLPB14] Hamoun Ghanbari, Marin Litoiu, Przemyslaw Pawluk, and Cornel Barna. Replica placement in cloud through simple stochastic model predictive control. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pages 80–87, June 2014.

[GMW00] David Garlan, Robert T Monroe, and David Wile. Acme: Architectural description of component-based systems. *Foundations of component-based systems*, 68:47–68, 2000.

[GMW10] David Garlan, Robert Monroe, and David Wile. Acme: an architecture description interchange language. In *CASCON First Decade High Impact Papers*, pages 159–173. IBM Corp., 2010.

[GSB+08] Heather J. Goldsby, Pete Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Danny Hughes. Goal-Based Modeling of Dynamically Adaptive System Requirements. In *Proc. of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 36–45. IEEE, 2008.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[HDPT04] Joseph L Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M Tilbury. *Feedback control of computing systems.* John Wiley & Sons, 2004.

[HJS$^+$14a] Edward Nicholas Hartley, Juan Luis Jerez, Andrea Suardi, Jan M Maciejowski, Eric C Kerrigan, and George A Constantinides. Predictive control using an fpga with application to aircraft control. *Control Systems Technology, IEEE Transactions on*, 22(3):1006–1017, 2014.

[HJS$^+$14b] E.N. Hartley, J.L. Jerez, A. Suardi, J.M. Maciejowski, E.C. Kerrigan, and G.A. Constantinides. Predictive control using an fpga with application to aircraft control. *Control Systems Technology, IEEE Transactions on*, 22(3):1006–1017, May 2014.

[HM08] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Comput. Surv.*, 40(3):7:1–7:28, August 2008.

[Hor01] Paul Horn. Autonomic computing: Ibm\'s perspective on the state of information technology. 2001.

[HSD10] Gabriel Hermosillo, Lionel Seinturier, and Laurence Duchien. Creating context-adaptive business processes. In *Service-Oriented Computing*, pages 228–242. Springer, 2010.

[ICG+04] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, and Jaime R Silva. Documenting component and connector views with uml 2.0. Technical report, DTIC Document, 2004.

[Ise82] H Isermann. Linear lexicographic optimization. *Operations-Research-Spektrum*, 4(4):223–228, 1982.

[JBEM14] Ivan J. Jureta, Alexander Borgida, Neil A. Ernst, and John Mylopoulos. The requirements problem for adaptive systems. *ACM Trans. Manage. Inf. Syst.*, 5(3):17:1–17:33, September 2014.

[JGR+14] J.L. Jerez, P.J. Goulart, S. Richter, G.A. Constantinides, E.C. Kerrigan, and M. Morari. Embedded online optimization for model predictive control at megahertz rates. *Automatic Control, IEEE Transactions on*, 59(12):3238–3251, Dec 2014.

[JKC12] Juan L. Jerez, Eric C. Kerrigan, and George A. Constantinides. A sparse and condensed QP formulation for predictive control of LTI systems. *Automatica*, 48(5):999–1002, 2012.

[Jur06] Matjaz B. Juric. *Business Process Execution Language for Web Services BPEL and BPEL4WS 2Nd Edition.* Packt Publishing, 2006.

[KC03] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[KKH+09] Dara Kusic, Jeffrey O Kephart, James E Hanson, Nagarajan Kandasamy, and Guofei Jiang. Power and perfor-

mance management of virtualized computing environments via lookahead control. *Cluster computing*, 12(1):1–15, 2009.

[KM90] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, November 1990.

[KOS06] Philippe Kruchten, Henk Obbink, and Judith Stafford. The past, present, and future for software architecture. *IEEE Softw.*, 23(2):22–30, March 2006.

[KR97] Joachim Karlsson and Kevin Ryan. A cost-value approach for prioritizing requirements. *IEEE Softw.*, 14(5):67–74, September 1997.

[KSSA09] Michiel Koning, Chang-ai Sun, Marco Sinnema, and Paris Avgeriou. Vxbpel: Supporting variability for web services in bpel. *Inf. Softw. Technol.*, 51(2):258–269, February 2009.

[KWR98] Joachim Karlsson, Claes Wohlin, and Björn Regnell. An evaluation of methods for prioritizing software requirements. *Information and Software Technology*, 39(14):939–947, 1998.

[Lap08] Jean-Claude Laprie. From dependability to resilience. In *38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks*, pages G8–G9. Citeseer, 2008.

[LDM95] A. Van Lamsweerde, R. Darimont, and P. Massonet. Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. In *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on*, pages 194–203, Mar 1995.

[Lee08] Edward A Lee. Cyber physical systems: Design challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369. IEEE, 2008.

[Lju99] Lennart Ljung. *System Identification: Theory for the User*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[Lju10] Lennart Ljung. Approaches to identification of nonlinear systems. In *Control Conference (CCC), 2010 29th Chinese*, pages 1–5, July 2010.

[LMPT13] Alberto Leva, Martina Maggio, Alessandro Vittorio Papadopoulos, and Federico Terraneo. *Control-Based Operating System Design*, volume 89 of *IET Control Engineering Series*. Institution of Engineering and Technology IET, 2013.

[LYM07] Alexei Lapouchnian, Yijun Yu, and John Mylopoulos. Requirements-driven design and configuration management of business processes. In *Business Process Management*, pages 246–261. Springer, 2007.

[Mac02] Jan M. Maciejowski. *Predictive Control: With Constraints*. Prentice Hall, 2002.

[MCN92] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *Software Engineering, IEEE Transactions on*, 18(6):483–497, 1992.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In

Software Engineering—ESEC'95, pages 137–153. Springer, 1995.

[MGMS11] Daniel A Menascé, Hassan Gomaa, Sam Malek, and Joao P Sousa. Sassy: A framework for self-architecting service-oriented systems. *Software, IEEE*, 28(6):78–85, 2011.

[MHP+12] Martina Maggio, Henry Hoffmann, Alessandro Vittorio Papadopoulos, Jacopo Panerati, Marco Domenico Santambrogio, Anant Agarwal, and Alberto Leva. Comparison of decision making strategies for self-optimization in autonomic computing systems. *ACM Transactions on Autonomous and Adaptive Systems*, 7(4):36:1–36:32, 2012.

[Mic] Microsoft Dynamic Systems Initiative, white paper, Microsoft, 2003.

[MPP09] Mirko Morandini, Loris Penserini, and Anna Perini. Operational Semantics of Goal Models in Adaptive Agents. In *Proc. of the 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 129–136. ACM, 2009.

[MTWJ96] Nenad Medvidovic, Richard N Taylor, and E James Whitehead Jr. Formal modeling of software architectures at multiple levels of abstraction. *ejw*, 714:824–2776, 1996.

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[NSGM16a] Chi Mai Nguyen, Roberto Sebastiani, Paolo Giorgini, and John Mylopoulos. Multi object reasoning with constrained goal model. *CoRR*, abs/1601.07409, 2016.

[NSGM16b] Chi Mai Nguyen, Roberto Sebastiani, Paolo Giorgini, and John Mylopoulos. Multi object reasoning with constrained goal model. *arXiv preprint arXiv:1601.07409*, 2016.

[Nus01] Bashar Nuseibeh. Weaving together requirements and architectures. *Computer*, 34(3):115–119, 2001.

[OGT+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and their Applications*, 14(3):54–62, May 1999.

[PCM+14] João Pimentel, Jaelson Castro, John Mylopoulos, Konstantinos Angelopoulos, and Vítor E. Silva Souza. From requirements to statecharts via design refinement. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 995–1000, New York, NY, USA, 2014. ACM.

[PCYZ10] Xin Peng, Bihuan Chen, Yijun Yu, and Wenyun Zhao. Self-tuning of software systems through goal-based feedback loop control. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 104–107, Sept 2010.

[PGH+01] Sujay Parekh, Neha Gandhi, Joe Hellerstein, Dawn Tilbury, T Jayram, and Joe Bigus. Using control theory to achieve service level objectives in performance management. In *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, pages 841–854, 2001.

[PLC+12] João Pimentel, Márcia Lucena, Jaelson Castro, Carla Silva, Emanuel Santos, and Fernanda Alencar. Deriving software architectural models from requirements models for adaptive systems: the stream-a approach. *Requirements Engineering*, 17(4):259–281, 2012.

[PMTL15] Alessandro Vittorio Papadopoulos, Martina Maggio, Federico Terraneo, and Alberto Leva. A dynamic modelling framework for control-based computing system design. *Mathematical and Computer Modelling of Dynamical Systems*, 21(3):251–271, 2015.

[QB03] S Joe Qin and Thomas A Badgwell. A survey of industrial model predictive control technology. *Control engineering practice*, 11(7):733–764, 2003.

[QP10] Nauman A. Qureshi and Anna Perini. Requirements Engineering for Adaptive Service Based Applications. In *Proc. of the 18th IEEE International Requirements Engineering Conference*, pages 108–111. IEEE, 2010.

[RL05] Paul Robertson and Robert Laddaga. Model based diagnosis and contexts in self adaptive software. In *Self-star Properties in Complex Information Systems*, pages 112–127. Springer, 2005.

[Rob07] William Robinson. Extended ocl for goal monitoring. *Electronic Communications of the EASST*, 9, 2007.

[RS79] D. T. Ross and K. E. Schoman, Jr. Classics in software engineering. chapter Structured Analysis for Requirements Definition, pages 363–386. Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[Saa80] T Saaty. Ahp: The analytic hierarchy process, 1980.

[SBP+08] João Pedro Sousa, Rajesh Krishna Balan, Vahe Poladian, David Garlan, and Mahadev Satyanarayanan. A software infrastructure for user–guided quality–of–service tradeoffs. In *Software and Data Technologies*, pages 48–61. Springer, 2008.

[SBW+10] Pete Sawyer, Nelly Bencomo, Jon Whittle, Emmanuel Letier, and Anthony Finkelstein. Requirements-aware systems: A research agenda for re for self-adaptive systems. In *Proceedings of the 2010 18th IEEE International Requirements Engineering Conference*, RE '10, pages 95–103, Washington, DC, USA, 2010. IEEE Computer Society.

[Sca09] Riccardo Scattolini. Architectures for distributed and hierarchical model predictive control–a review. *Journal of Process Control*, 19(5):723–731, 2009.

[SG02] Bradley Schmerl and David Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, SEKE '02, pages 241–248, New York, NY, USA, 2002. ACM.

[SHMK08] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. From goals to components: A combined approach to self-management. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, pages 1–8, New York, NY, USA, 2008. ACM.

[SHMK10] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. Exploiting non-functional preferences in architectural adaptation for self-managed systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 431–438, New York, NY, USA, 2010. ACM.

[SLAM13] Vítor E. Silva Souza, Alexei Lapouchnian, Konstantinos Angelopoulos, and John Mylopoulos. Requirements-driven software evolution. *Computer Science - R&D*, 28(4):311–329, 2013.

[SLM11] Vítor Estêvão Silva Souza, Alexei Lapouchnian, and John Mylopoulos. System identification for adaptive software systems: A requirements engineering perspective. In *Conceptual Modeling - ER 2011, 30th International Conference, ER. Proceedings*, pages 346–361, 2011.

[SLM12a] Vítor E. Silva Souza, Alexei Lapouchnian, and John Mylopoulos. *On the Move to Meaningful Internet Systems: OTM 2012: Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2012, Rome, Italy, September 10-14, 2012. Proceedings, Part I*, chapter Requirements-Driven Qualitative Adaptation, pages 342–361. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[SLM12b] VítorE.Silva Souza, Alexei Lapouchnian, and John Mylopoulos. Requirements-driven qualitative adaptation. In Robert Meersman, Hervé Panetto, Tharam Dillon, Stefanie Rinderle-Ma, Peter Dadam, Xiaofang Zhou, Siani Pearson, Alois Ferscha, Sonia Bergamaschi, and IsabelF. Cruz, editors, *On the Move to Meaningful Internet Systems: OTM*

2012, volume 7565 of *Lecture Notes in Computer Science*, pages 342–361. Springer Berlin Heidelberg, 2012.

[SLRM11] Vítor Estêvão Silva Souza, Alexei Lapouchnian, William N. Robinson, and John Mylopoulos. Awareness requirements for adaptive systems. In *2011 ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*, pages 60–69, 2011.

[SMK11] Daniel Sykes, Jeff Magee, and Jeff Kramer. Flashmob: Distributed adaptive self-assembly. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 100–109, New York, NY, USA, 2011. ACM.

[SP07] Sigurd Skogestad and Ian Postlethwaite. *Multivariable feedback control: analysis and design*, volume 2. Wiley New York, 2007.

[SS12] Vítor Estêvão Silva Souza. *Requirements-based Software System Adaptation*. PhD thesis, University of Trento, 2012.

[ST15] Roberto Sebastiani and Patrick Trentin. Optimathsat: A tool for optimization modulo theories. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 447–454, 2015.

[Sun] Sun Microsystems: Sun N1 Service Provisioning System (2007). Accessed: 2016-02-23.

[TM06] Vijay Tewari and Milan Milenkovic. Standards for autonomic computing. *Intel technology journal*, 10(4), 2006.

[vL00] Axel van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 5–19, New York, NY, USA, 2000. ACM.

[vOvdLKM00] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, March 2000.

[WB10] Yang Wang and S. Boyd. Fast model predictive control using online optimization. *Control Systems Technology, IEEE Transactions on*, 18(2):267–278, March 2010.

[Wei93] Mark Weiser. Some computer science issues in ubiquitous computing. *Commun. ACM*, 36(7):75–84, July 1993.

[Whi04] Stephen A White. Introduction to bpmn. *IBM Cooperation*, 2(0):0, 2004.

[WSB+10] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty Cheng, and Jean-Michel Bruel. RELAX: a language to address uncertainty in self-adaptive systems requirement. *Requirements Engineering*, 15(2):177–196, 2010.

[YGMM11] Eric S. K. Yu, Paolo Giorgini, Neil Maiden, and John Mylopoulos. *Social Modeling for Requirements Engineering.* MIT Press, 1st edition, 2011.

[YLL+08] Yijun Yu, Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Julio CSP Leite. From goals to high-variability software design. In *Foundations of Intelligent Systems*, pages 1–16. Springer, 2008.

[ZCB10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[ZJ97] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, January 1997.

[ZRD+14] Melanie N. Zeilinger, Davide M. Raimondo, Alexander Domahidi, Manfred Morari, and Colin N. Jones. On real-time robust model predictive control. *Automatica*, 50(3):683–694, 2014.

[ZSL14] Parisa Zoghi, Mark Shtern, and Marin Litoiu. Designing search based adaptive systems: a quantitative approach. In *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, Proceedings*, pages 7–16, 2014.